Высшее профессиональное образование

БАКАЛАВРИАТ

АРХИТЕКТУРА ИНФОРМАЦИОННЫХ СИСТЕМ

Учебник

Допущено
Учебно-методическим объединением вузов
по университетскому политехническому образованию
в качестве учебника для студентов
высших учебных заведений,
обучающихся по направлению подготовки
230400 «Информационные системы и технологии»

Москва
Издательский центр «Академия»
2012

УДК 681.518(075.8)
ББК 65.39я73
А878

Рецензенты:

зав. кафедрой «Интеллектуальные техиологии и системы» Московского государственного технического университета радиотехиики, электроники и автоматики, академик РАЕП, д-р физ.-мат. наук, проф. *В. В. Нечаев*;

зав. кафедрой «Автоматизпрованные системы унравления» Московского государственного горного университета, заслуженный работник Высшей школы Российской Федерации, д-р техи. наук, проф. *Н. И. Федунец*

А878 **Архитектура** информационных систем: учебник для студ. учреждений высш. проф. образования / Б. Я. Советов, А. И. Водяхо, В. А. Дубенецкий, В. В. Цехановский. — М. : Издательский центр «Академия», 2012. — 288 с. — (Сер. Бакалавриат).

ISBN 978-5-7695-8827-3

Учебник создан в соответствии с Федеральным государственным образовательным стандартом но направлению нодготовки 230400 «Информационные системы и техиологии» (квалификация «бакалавр»).

Дана характеристика эволюции приложений и нлатформенных техиологий, приведена классификация информационных систем и моделей их нредставления, нодробно рассмотрены проблемы концентуального моделирования информационных систем и существующие архитектурные стили их проектпрования.

С нозиций наконленного отечественного и зарубежного оныта рассмотрены вопросы решения задач нроектпрования информационных систем с иснользованием наттернов и каркасов, комнонентной техиологии, сервисно-ориентированных технологий, нортальных техиологий реализации информационных систем. Приведены примеры архитектурных решений, взятых из практики проектпрования информационных систем.

Для студентов учреждений высшего нрофессионального образования.

УДК 681.518(075.8)
ББК 65.39я73

Оригинал-макет данного издания является собственностью Издательского центра «Академия», и его воспроизведение любым способом без согласия правообладателя запрещается

ISBN 978-5-7695-8827-3

© Советов Б.Я., Водяхо А. И., Дубенецкий В.А., Цехановский В.В., 2012
© Образовательно-издательский центр «Академия», 2012
© Оформление. Издательский центр «Академия», 2012

ВВЕДЕНИЕ

Информационные технологии являются не только объектом исследований и разработки, но и средством создания информационных систем в различных предметных областях. Песмотря на специфику конкретных объектов, удалось разработать методологию, модели, методы и средства прикладных информационных технологий, что позволяет снизить затраты и сократить сроки информатизации.

Практическое использование информационных технологий тесно связано с вопросами маркетинга и менеджмента информационных ресурсов, технологий и услуг, методологией проектирования информационных систем, управления качеством и стандартизации информационных технологий. В настоящее время в целом сформировалась идеология и практика применения информационных технологий. Разнообразие задач, решаемых с помощью информационных систем (ИС), привело к появлению множества разнотипных систем, различающихся принципами построения и заложенными в них правилами обработки информации. Поэтому возникла необходимость организации информационных процессов и технологий с использованием системного подхода в основу которого положена архитектура информационных систем.

В настоящем учебнике отражены современные достижения в области проектирования информационных систем на основе архитектурных решений, что обеспечивает переход к промышленным методам и средствам работы с информацией в различных сферах человеческой деятельности. Это дает возможность расширения существующих методов и средств реализации информационных систем и перехода к принципиально новым, основанным на высоких технологиях.

В российской высшей школе подготовка дипломированных специалистов — профессионалов в области информационных технологий осуществляется по ряду направлений, из которых наиболее глубоко реализуется по направлению «Информационные системы и технологии». Студентам, обучающимся по этому направлению, в основном и предназначается данный учебник, включающий в себя семь глав основного материала.

Первая глава посвящена общей характеристке архитектурного подхода к информационным системам. Приведены основные понятия и определения, дана характеристика информационной системы как объекта архитектуры, показана взаимосвязь архитектуры и про-

цесса проектирования ИС, последовательно раскрывается эволюция платформенных архитектур ИС.

Вторая глава посвящена архитектурным стилям, используемым в ИС. Приводится понятие архитектурного стиля и классификация архитектурных стилей. Последовательно рассматриваются различные архитектрные стили: потоки данных, вызов с возвратом, независимые компоненты, централизованные данные, системы, работающие по принципу виртуальной машины.

В третьей главе детально раскрываются паттерны, антипаттерны и фреймворки в архитектуре ИС, рассматриваются примеры использования.

В четвертой главе раскрываются компонентные технологии реализации информационных систем. Определяются основные понятия компонента, компонентных технологий, квазикомпонентно-ориентированных технологий. Дается подробная характеристика технологий, основанных на объектной модели компонентов COM+, .NET. Значительное внимание уделено широко используемым технологиям CORBA и Enterprise JavaBeans (EJB).

В пятой главе авторами приводится анализ сервисно-ориентированных технологий реализации информационных систем прикладных информационных технологий, в состав которых включены: сервисно-ориентированные архитектуры (СОА) и Web-сервисы, язык XML при работе с Web-сервисами, WSDL-описание, UDDI реестр, бизнес-реестр ebXML, язык WS-Inspection для поиска Web-служб, спецификации WS-*.

Непосредственно применению архитектурного подхода при проектировании ИС посвящена шестая глава книги. Последовательно рассмотрены общие принципы организации взаимодействий в ИС; системы, ориентированные на работу с сообщениями; язык описания бизнес-процессов BPEL; бизнес-правила; порталы и портлеты; корпоративные сервисные шины, сервисно-ориентированная архитектура и сервисно-ориентированная организация.

В седьмой главе рассмотрены вопросы практического применения архитектурных решений для разработки приложений на примере корпоративных информационных систем (КИС). Основное внимание уделено архитектурным решениям на основе каркасов. Подробно рассмотрены базовые каркасы для моделирования структуры классов и их свойств и каркасы для поддержки функций приложения.

Контрольные вопросы, приведенные в конце каждой главы, дают возможность студенту проверить качество усвоения материала.

Авторы надеются, что данный учебник будет способствовать повышению качества подготовки бакалавров, а также будет полезен всем читателям, интересующимся современным состоянием и перспективами развития информационных систем и технологий.

Глава 1

АРХИТЕКТУРНЫЙ ПОДХОД К ИНФОРМАЦИОННЫМ СИСТЕМАМ

1.1. Основные понятия и определения

Современные ИС и информационные технологии (ИТ) достигли такого уровня развития, когда на первый план выходит бизнес-оценка проектов, а не личные пристрастия разработчиков или заказчиков. В связи с этим большое внимание в настоящее время уделяется архитектуре информационных систем.

Термин «архитектура» в применении к ИС уже давно стал привычным, так как грамотное построение информационной системы, эффективно и надежно функционирующей, является не менее сложной задачей, чем проектирование и возведение современного многофункционального здания.

Архитектура (лат. *architectura*) — искусство проектировать и строить здания и другие сооружения (комплексы), создающие материально организованную среду, необходимую людям для их жизни и деятельности, в соответствии с современными техническими возможностями и эстетическими воззрениями общества. Постепенно классическое определение архитектуры трансформировалось в применении к техническим системам как принципиальное устройство чего-либо сложного, общий вид, вид без указания конкретных инженерных расчетов.

Когда речь заходит об «архитектуре информационной системы», обычно не возникает недостатка в определениях. На сайте SEI (Software Engineering Institute) имеется специальный раздел, посвященный определениям архитектуры, где их собрано более ста [12].

Рассмотрим определение «архитектуры информационной системы», которое дают различные источники:

- архитектура — организационная структура системы;
- архитектура информационной системы — концепция, определяющая модель, структуру, выполняемые функции и взаимосвязь компонентов информационной системы;
- архитектура — базовая организация системы, воплощенная в ее компонентах, их отношениях между собой и окружением, а также принципы, определяющие проектирование и развитие системы;

• архитектура — набор значимых решений по поводу организации системы программного обеспечения, набор структурных элементов и их интерфейсов, при помощи которых компонуется система вместе с их поведением, определяемым во взаимодействии между этими элементами, компоновка элементов в постепенно укрупняющиеся подсистемы, а также стиль архитектуры, который направляет эту организацию (элементы и их интерфейсы, взаимодействия и компоновку);

• архитектура программы или компьютерной системы — структура или структуры системы, которые включают элементы программы, видимые извне свойства этих элементов и связи между ними;

• архитектура — структура организации и связанное с ней поведение системы. Архитектуру можно рекурсивно разобрать на части, взаимодействующие посредством интерфейсов, связи, которые соединяют части и условия сборки частей. Части, взаимодействующие через интерфейсы, включают классы, компоненты и подсистемы;

• архитектура программного обеспечения системы или набора систем состоит из всех важных проектных решений в отношении структур программы и взаимодействий между этими структурами, составляющих системы. Проектные решения обеспечивают желаемый набор свойств, которые должна поддерживать система, чтобы быть успешной. Проектные решения предоставляют концептуальную основу для разработки системы, ее поддержки и обслуживания.

Для того чтобы разобраться в этом многообразии определений, выделим наиболее существенные стороны архитектуры ИС.

Основным критерием выбора архитектуры и инфраструктуры ИС в условиях рыночной экономики является минимизация совокупной стоимости владения системой. Из этого следуют два основных тезиса.

1. В проектах построения информационных систем, для которых важен экономический эффект, необходимо выбирать архитектуру системы с минимальной совокупной стоимостью владения.

2. Совокупная стоимость владения ИС состоит из плановых затрат и стоимости рисков. Стоимость рисков определяется стоимостью бизнес-рисков, вероятностями технических рисков и матрицей соответствия между ними. Матрица соответствия определяется архитектурой ИС.

Термин «архитектура информационной системы» обычно довольно согласованно понимается специалистами на уровне подсознания, но при этом и определения этого понятия неоднозначны. Имеются два основных класса определений архитектур — «идеологические» и «конструктивные».

Основные идеологические определения архитектуры ИС таковы:

1. Архитектура ИС — набор решений, наиболее существенным образом влияющих на совокупную стоимость владения системой;

2. Архитектура ИС — набор ключевых решений, неизменных при изменении бизнес-технологии в рамках бизнес-видения.

Эти определения объединяет то, что если ключевое решение приходится изменять при изменении бизнес-технологии в рамках бизнес-видения, то резко возрастает стоимость владения системой. Как следствие возникает понимание важности принятия архитектуры системы как стандарта предприятия ввиду значимости архитектурных решений и их устойчивости по отношению к изменениям бизнес-технологии. Важный вывод из первого определения — архитектура системы должна строиться еще на стадии технико-экономического обоснования системы.

Выделим несколько наиболее значимых и принципиально различных типов рисков:

- проектные риски при создании системы;
- технические риски, состоящие в простоях, отказах, потере или искажении данных;
- риски бизнес-потерь, связанные с эксплуатацией системы (возникающие, в конечном счете, из-за технических рисков). Такие риски бизнес-потерь назовем *бизнес-рисками*. Каждый бизнес-риск принадлежит, по крайней мере, одному из вариантов бизнес-использования (business use case) системы. Например, для интернет-магазина бизнес-риски «Покупатель не может сделать заказ и уходит» и «Покупатель делает заказ, но уходит рассерженный функционированием системы» принадлежат вариантам бизнес-использования «Сделать заказ»;
- риски бизнес-потерь, связанные с вариативностью бизнес-процессов. При этом потери происходят оттого, что, во-первых, бизнес-процессы надо изменять, а ИС не готова к этому, и потери связаны с неоптимальным функционированием бизнеса; во-вторых, приходится платить за модификацию системы. Такие риски бизнес-потерь назовем *неопределенностями* (RUP упоминает аналогичные по смыслу «варианты изменения», change cases).

Затраты на создание и эксплуатацию системы с некоторой точностью оцениваются достаточно стандартно. Динамические бизнес-риски количественно учесть невозможно, и их следует оценивать исключительно качественно (на уровне понимания, насколько бизнес-процессы в организации являются определенными, застывшими или неустоявшимися). Наиболее интересная часть совокупной стоимости владения системой — статические бизнес-риски и риски разработки.

Каждый вариант бизнес-использования реализуется с помощью набора операций соответствующих бизнес-процессов, а бизнес-риск возникает по причине неисполнения одной или нескольких операций. Такие риски мы называем *операционными*. Таким образом, операционные риски являются источниками бизнес-рисков.

Например, источником риска «Покупатель не может сделать заказ и уходит» может быть операционный риск «Информация о заказе не может быть передана для обработки в систему».

В свою очередь операционные риски для автоматизированных операций могут возникать по причине технических рисков. Например, операционный риск «Информация о заказе не может быть передана для обработки в систему» может возникнуть по причине реализации технического риска «Обрыв канала связи».

Кроме того, бизнес-риск может быть парирован соответствующей организацией процесса и/или архитектурным решением.

Конструктивно архитектура обычно определяется как набор ответов на следующие вопросы:

- что делает система?;
- на какие части она разделяется?;
- как эти части взаимодействуют?;
- где эти части размещены?.

Таким образом, архитектура ИС является логическим построением, или моделью, и влияет на совокупную стоимость владения через набор связанных с ней решений по выбору средств реализации, СУБД, операционной платформы, телекоммуникационных средств и т. п. — т. е. через то, что мы называем *инфраструктурой* ИС. Еще раз подчеркнем, что инфраструктура включает решения не только по программному обеспечению, но и по аппаратному комплексу и организационному обеспечению. Это вполне соответствует пониманию системы в наиболее современных стандартах типа ISO/IEC 15288 [14].

1.2. Характеристика информационной системы как объекта архитектуры

В настоящее время не существует единого устоявшегося и общепринятого определения понятия «информационная система». Вызывает сомнение, что такое определение когда-нибудь появится.

Дело в том, что информационные системы достаточно часто имеют сложную структуру и ее адекватное описание возможно только при использовании нескольких точек зрения.

В качестве официального определения данного понятия можно рассматривать определение, которое дает Федеральный закон Российской Федерации от 27 июля 2006 г. № 149-ФЗ «Об информации, информационных технологиях и о защите информации»: «Информационная система — совокупность содержащейся в базах данных информации и обеспечивающих ее обработку информационных технологий и технических средств».

Это достаточно общее определение, причем обращает на себя внимание тот факт, что во главу угла ставится понятие базы данных.

Другие известные определения понятия «информационная система» часто носят еще более общий характер.

Например, редакционная коллегия журнала «Information Systems», выпускаемом английским издательством Pergamon Press, определяет

информационные системы как «аппаратно-программные системы, которые поддерживают приложения с интенсивной обработкой данных (Data-Intensive Applications)».

Ряд авторов предлагает использовать два определения. В узком смысле ИС определяют как подмножество компонентов, включающее базы данных и приложения, т.е. ИС в узком смысле определяют как программно-аппаратную систему, предназначенную для автоматизации целенаправленной деятельности конечных пользователей и обеспечивающую, в соответствии с заложенной в нее логикой обработки, возможность получения, модификации и хранения информации [8].

Еще более общую трактовку понятия ИС можно найти в [7], где дается следующее определение ИС: «информационной системой называется комплекс, включающий вычислительное и коммуникационное оборудование, программное обеспечение, лингвистические средства и информационные ресурсы, а также системный персонал, обеспечивающий поддержку динамической информационной модели некоторой части реального мира для удовлетворения информационных потребностей пользователей».

Последнее определение заслуживает отдельного внимания, поскольку в нем приведены практически все возможные точки зрения на ИС, хотя это не означает, что применительно к данной конкретной ИС действительно необходимо использовать все перечисленные точки зрения на ИС.

В ИТ индустрии термин «архитектура» используется достаточно часто. Данный термин может относиться к организации, программно-аппаратному комплексу, программной системе или центральному процессору.

Применительно к организации обычно используют понятие корпоративная архитектура (enterprise architecture), при этом выделяются следующие типы архитектур: бизнес-архитектура (Business architecture), ИТ-архитектура (Information Technology Architecture), архитектура данных (Data Architecture), архитектура приложения (Application Architecture) или программная архитектура (Software Architecture), техническая архитектура (Hardware Architecture). Совокупность данных архитектур и является архитектурой ИС (рис. 1.1).

Бизнес-архитектура, или архитектура уровня бизнес-процессов, определяет бизнес-стратегии, управление, организацию, ключевые бизнес-процессы в масштабе предприятия, причем не все бизнес-процессы реализуются средствами ИТ-технологий. Бизнес-архитектура отображается на ИТ-архитектуру.

ИТ-архитектура рассматривается в трех аспектах:

- обеспечивает достижение бизнес-целей посредством использования программной инфраструктуры, ориентированной на реализацию наиболее важных бизнес-приложений;
- среда, обеспечивающая реализацию бизнес-приложений;

Рис. 1.1. Архитектура информационной системы

• совокупность программных и аппаратных средств, составляющая информационную систему организации и включающая, в частности, базы данных и промежуточное программное обеспечение.

Архитектура данных организации включает логические и физические хранилища данных и средства управления данными. Архитектура данных должна быть поддержана ИТ-архитектурой. В современных ИТ-системах, ориентированных на работу со знаниями, иногда выделяют отдельный тип архитектуры — архитектуру знаний (Knowledge Architecture).

Ирограммная архитектура отображает совокупность программных приложений. Программное приложение — это компьютерная программа, ориентированная на решение задач конечного пользователя. Архитектура приложения — это описание отдельного приложения, работающего в составе ИТ-системы, включая его программные интерфейсы. Архитектура приложения базируется на ИТ-архитектуре и использует сервисы, предоставляемые ИТ-архитектурой.

Техническая архитектура характеризует аппаратные средства и включает такие элементы, как процессор, память, жесткие диски, периферийные устройства, элементы для их соединения, а также сетевые средства.

Часто нельзя провести четкой границы между ИТ-архитектурой и архитектурой отдельного приложения. В частности в том случае, когда некоторую фупкцию требуется реализовать в нескольких приложений, она может быть перенесена на уровень ИТ-архитектуры. Обычно приложения интегрируются средствами ИТ-архитектуры. Элементы архитектуры данных часто интегрируются в приложения.

Ввиду многообразия ИС остановимся на их классификации. В последние годы все более широкое распространение получил доменный подход к описанию ИТ-архитектур. Под доменной архитектурой понимают эталонную модель, описывающую множество систем, которые реализуют похожую структуру, функциональность и поведение.

Доменную архитектуру можно рассматривать как метамодель, описывающую множество решений.

Схемы классификации архитектур ИС, основанные на доменном подходе, показаны на рис. 1.2 и 1.3. Па верхнем уровне выделяются два типа доменов: домены задач (Problem domains) (см. рис. 1.2) и домены решений (Solution Domains) (см. рис. 1.3).

Можно выделить следующие основные характеристики домена задач:

- характер решаемых задач;
- тип домена;
- предметная область;
- степень автоматизации;
- масштаб применения.

По характеру обработки данных ИС делятся следующим образом:

- на системы, ориентированные на решение крупномасштабных задач преимущественно вычислительного характера;
- информационно-справочные (информационно-поисковые) ИС, в которых нет сложных алгоритмов обработки данных, а целью системы является поиск и выдача информации в удобном для пользователя виде;

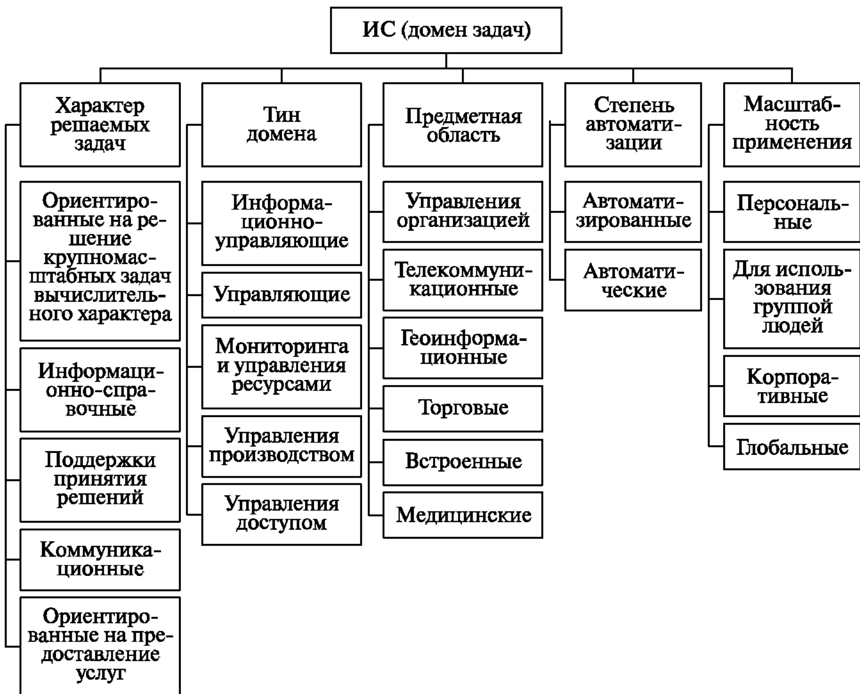


Рис. 1.2. Классификация архитектур ИС, основанная на домене задач

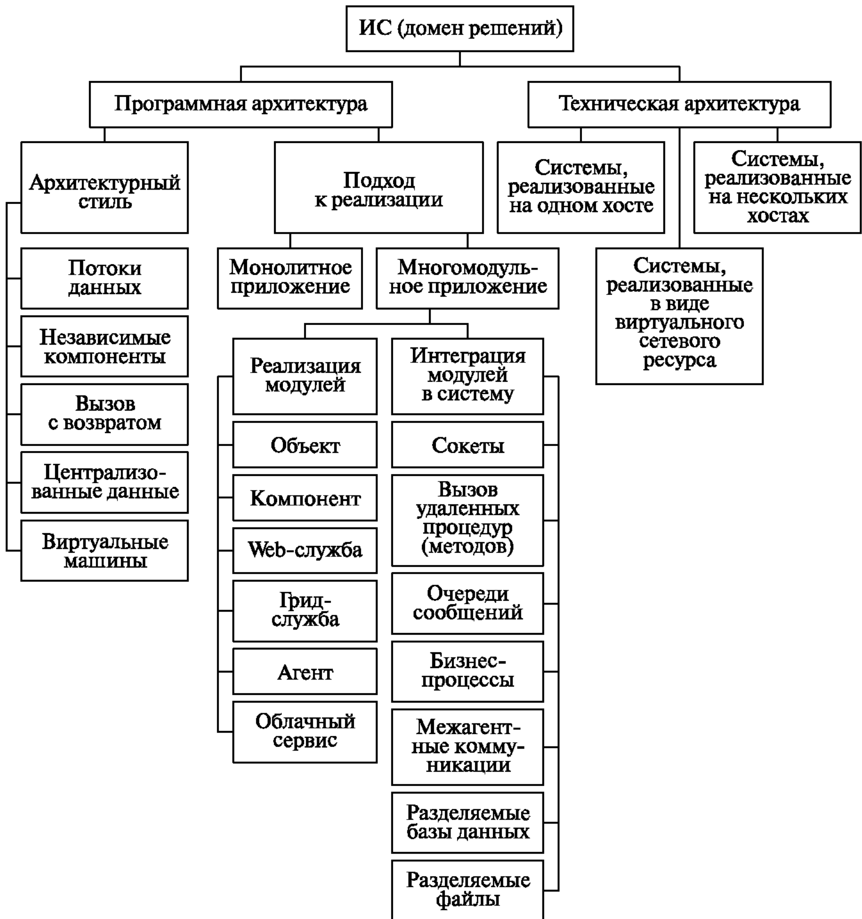


Рис. 1.3. Классификация архитектур ИС, основанная на домене решений

- системы поддержки принятия решений;
- коммуникационные системы;
- ИС, ориентированные на предоставление услуг (сервисов), таких как доступ в Интернет, сервисы хранения данных, доступа к вычислительным ресурсам, доступа к данным и т. п.

По принадлежности к базовому домену можно выделить следующие базовые домены задач [36]: информационно-управляющие системы — ИУС (Management Information Systems), управляющие системы — УС (Process Control Systems), системы мониторинга и управления ресурсами — СМУР (Resource Allocation and Tracking Systems), системы управления производством — СУП (Manufacturing Systems), системы управления доступом — СУД (Access Control Systems).

Ио принадлежности к предметной области обычно ИС ориентированы на использование и удовлетворение информационных потребностей в рамках конкретной предметной области. В настоящее время ИС используются практически повсеместно, перечислить все области, в которых используются ИС, просто невозможно. В качестве примера можно указать следующие области, в которых ИС активно используются:

- системы управления организацией — ИС, предназначенные для выполнения фупкций управления организацией (предприятием);
- телекоммупикационные системы — ИС, предназначенные для реализации фупкций, связанных передачей данных;
- геоинформационные системы — ИС, обеспечивающие сбор, хранение, обработку, доступ, отображение и распространение пространственно-координированных данных (пространственных данных);
- торговые ИС;
- встроенные системы управления сложными объектами, такими как самолеты и корабли;
- медицинская информационная система — ИС, предназначенные для использования в лечебных учреждениях.

Ио степени автоматизации различают автоматизированные ИС (предполагают участие человека в ее фупкционировании) и автоматические ИС (функционируют без участия оператора).

Ио масштабности применения ИС делятся:

- на персональные — ИС, предназначенные для использования одним человеком;
- ИС, предназначенные для совместного использования группой людей, например сотрудниками одного подразделения;
- корпоративные — ИС, охватывающие информационные процессы отдельной организации;
- глобальные — ИС, охватывающие информационные процессы многих организаций.

Основными характеристиками домена решений являются программная и техническая архитектуры.

Применительно к уровню программной архитектуры выделим следующие характеристики: используемый архитектурный стиль и способ реализации.

Существуют пять групп архитектурных стилей: потоки данных, независимые компоненты, вызов с возвратом, централизованные данные, виртуальные машины. Более подробно архитектурные стили будут рассмотрены ниже.

Реализация программной архитектуры может быть осуществлена двумя альтернативными подходами: монолитное приложение, многомодульное приложение.

Основными характеристиками многомодульных приложений являются способы реализации модулей, способ их интеграции в систему.

Основные подходы к реализации модулей:
- представление модуля как объекта;
- представление модуля как компонента;
- реализация модуля в виде Web-службы;
- реализация модуля в виде грид-службы;
- реализация модуля в виде агента;
- реализация модуля в виде облачного сервиса.

Основные подходы к интеграции модулей:
- сокеты;
- вызов удаленных процедур (методов);
- очереди сообщений;
- бизнес-процессы;
- межагентные коммуникации;
- разделяемые базы данных;
- разделяемые файлы.

Применительно к уровню технической архитектуры ИС можно разделить:
- на системы, реализованные на одном хосте;
- системы реализованные на нескольких хостах;
- системы, реализованные в виде виртуального сетевого ресурса.

Рассмотрим более подробно ИС, выделенные на основе доменного подхода.

К ИУС относят ИТ-системы, обеспечивающие выдачу консолидированных данных, которые могут быть использованы для поддержки принятия решений, и отчетов на основе данных из различных независимых источников. К УС относят ИТ-системы, обеспечивающие мониторинг и управление параметрами систем. К СМУР относят ИТ-системы, которые позволяют обрабатывать запросы от других подсистем, входящих в состав системы. К СУП относят ИТ-системы, ориентированные на поддержку процессов получения готовых продуктов из сырья. К СУП относят ИТ-системы, обеспечивающие контроль доступа к пассивным объектам со стороны активных подсистем. Они, по существу, являются подсистемами обеспечения безопасности.

Эти типы ИС различаются как используемыми архитектурными решениями, так и требованиями к различным типам ИС. В табл. 1.1 приведены требовования, предъявляемые к отдельным характеристикам рассматриваемых типов ИС [36].

Кроме перечисленных выше пяти базовых типов ИС можно выделить и другие типы ИС.

Очевидно, что реальные ИС значительно более сложные и являются композицией перечисленных выше типов ИС.

Информационно-управляющие системы (ИУС). Это достаточно известная эталонная модель, которая интенсивно используется, по крайней мере, с середины 1970-х гг. Отличительной особенностью данного класса систем является то, что реализуется процесс сбора

Таблица 1.1. **Требования к различным типам ИС**

Характеристика	ИУС	УС	СМУР	СУП	СУД
Функциональность	Высокие	Невысокие	Средние	Средние	Высокие
Надежность	Средние	Высокие	Невысокие	Средние	Невысокие
Эффективность	Средние	Высокие	Средние	Высокие	Средние
Удобство использования	Высокие	Средние	Средние	Невысокие	Средние
Удобство сопровождения	Средние	Средние	Средние	Средние	Средние
Переносимость	Средние	Высокие	Средние	Средние	Средние

данных, поступающих из разных источников, в частности от датчиков. Затем данные обрабатываются и выдаются различным категориям пользователей (stakeholders) в форме отчета, данные из отчета учитываются при принятии решения.

Подобные системы используются как в сфере административного управления, так и для управления техническими системами.

Обобщенная структура ИУС показана на рис. 1.4 и включает следующие компоненты:

• источники данных (ИД);

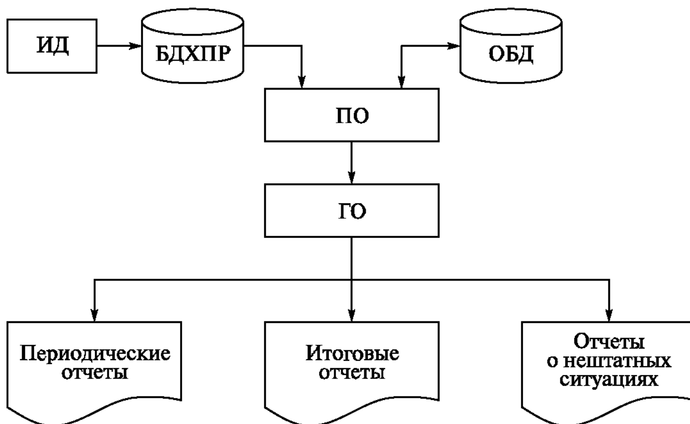


Рис. 1.4. Обобщенная структура ИУС

- основную базу данных (ОБД);
- базу данных для хранения промежуточных результатов (БДХПР);
- подсистему обработки(ПО);
- подсистему генерации отчетов (ГО);
- набор человеко-машинных интерфейсов.

Основная база данных предназначена для накопления данных на протяжении достаточно длительного промежутка времени. Содержимое ОБД данных регулярно пополняется. Данные обычно не удаляются.

ИД включают «сырые данные», с которыми работают транзакции; БДХПР предназначается преимущественно для работы с транзакциями, т.е. данная база хранит данные, относящиеся к выполняемым транзакциям. БДХПР хранит записи, которые появляются как результат обработки данных, поступающих от источников. Если во входных данных обнаруживается ошибка, то информация об ошибке заносится в лог-файл и формируется отчет о появлении ошибки.

ОБД хранит исторические данные и различную информацию об организации. Информация обычно обобщается на нескольких уровнях. Обычно такие данные хорошо структурированы и могут использоваться в качестве источников данных для систем поддержки принятия решений и других приложений, например data mining.

Пабор человеко-машинных интерфейсов позволяет вводить и модифицировать данные, используемые транзакциями. Данные интерфейсы дают возможность пользователям получать доступ к отчетам, генерируемым системой по запросу пользователей, на регулярной основе, либо при возникновении исключительных ситуаций. В общем случае интерфейсы используются для реализации оперативного, тактического и стратегического управления.

ПО предназначена для обеспечения работы с транзакциями, в частности для сравнения результатов выполнения транзакции с эталонными значениями, хранящимися в ОБД.

Подсистема генерации отчетов (ГО) обеспечивает представление информации в удобном для пользователя виде.

Данные в ИУС принято разделять на три основные группы: оперативные, тактически и стратегические. Оперативные данные формируются из поступающих от источников и используются при принятии решений, связанных с достижением краткосрочных целей.

Тактические данные формируются из оперативных и используются при принятия решений, связанных с достижением среднесрочных целей.

Стратегические данные являются обобщением данных тактического уровня. Стратегические данные, по существу, являются корпоративным знанием.

Управляющие системы (УС). Их называют еще системами управления процессами. Они представляют собой широкий класс систем, назначением которых является измерение некоторых параметров системы и выдача управляющих воздействий.

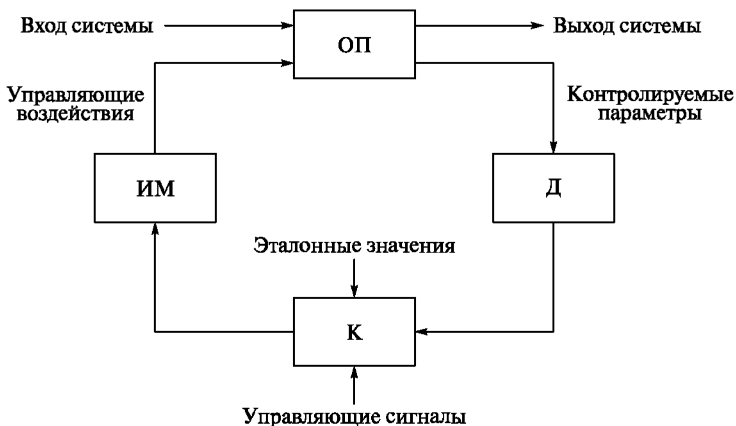


Рис. 1.5. Обобщенная структура УС

Следует заметить, что к системам управления процессами относят не только промышленные системы, но и любые другие системы, как реальные, так и виртуальные, в процессе фупкционирования которых реализуется мониторинг параметров и выдача управляющих воздействий.

УС широко и успешно используются в течение многих лет.

Одним из широко распространенных типов УС следует считать системы управления производственными процессами.

Основными элементами типовой УС (рис. 1.5) являются основной процесс (ОП), датчики (Д), исполнительные механизмы (ИМ), контроллер (К).

Типовая УС работает следующим образом. С помощью датчиков Д считывают некоторый набор параметров, характеризующий состояние ОП. Контроллер представляет собой аналоговое или цифровое устройство, реализующее требуемую фупкцию управления. В процессе работы на вход контроллера поступают также эталонные значения параметров. Контроллер выдает управляющие воздействия, которые поступают на вход ИМ.

Системы мониторинга и управления ресурсами (СМУР). Системы данного класса широко используются для решения задач управления производством, финансами и других видов бизнеса.

Отличительной особенностью данного класса систем является то, что требуется отслеживать состояние некоторой физической или логической сущности с момента ее поступления на вход системы до момента ее выхода из системы. В качестве такой сущности может выступать поток информации или физический объект. Под термином «мониторинг» в данном случае понимается выполнение действий, связанных с определением текущего состояния отслеживаемого объекта, в частности места нахождения.

Под термином «управление» понимается возможность изменения текущего состояния объекта, например изменения его местонахождения.

В качестве примеров СМУР можно привести следующие системы:

- системы управления складами;
- банковские системы;
- системы управления документооборотом;
- системы управления торговыми сетями;
- системы управления транспортными потоками;
- системы управления глобальными сетями.

Основное назначение систем данного типа — это управление потоком работ. При этом реализуются такие типовые функции, как регистрация информации и отслеживание ее изменений, перемещение в реальном или виртуальном мире и уничтожение. Кроме того, обычно имеется возможность назначать исполнителей для реализации отдельных активностей.

Системы управления производством (СУП). Термин «manufacture» появился впервые в 1622 г. как перевод с латинского *manu factum* (изготовленный вручную) и относился к созданию реальных вещей.

Отличительной особенностью систем, принадлежащих к данному классу, является то, что на вход системы поступает сырье, а на выходе образуется готовый продукт. При этом не имеет значения физическая природа производимого продукта. Это может быть либо реальный продукт, имеющий физическое воплощение, либо некоторая информация, относящаяся к физическим сущностям. Во всех случаях на входе производственной системы имеется полуфабрикат, который обрабатывается и превращается в полуфабрикат. Полуфабрикат может проходить одну или несколько стадий обработки, пока не превратится в готовый продукт, который удовлетворяет потребностям конкретных заинтересованных лиц, которым он поставляется.

Анализ СУП имеет ряд особенностей. Во-первых, рекомендуется максимально подробно документировать систему как с точки зрения входной и выходной информации, так и с точки зрения полуфабрикатов, получаемых на отдельных этапах производства. Во-вторых, алгоритмы, используемые для управления СУП, обычно отличаются высокой сложностью. В-третьих, поскольку СУП обычно предоставляют товары и сервисы другим системам, выходные данные должны выдаваться с учетом принятых стандартов.

Концептуальная модель функционирования СУП показана на рис. 1.6.

Практически во всех СУП можно выделить три основных аспекта:

- материальные потоки (преобразование сырья в конечный продукт);
- информационные потоки (планирование и управление производством);

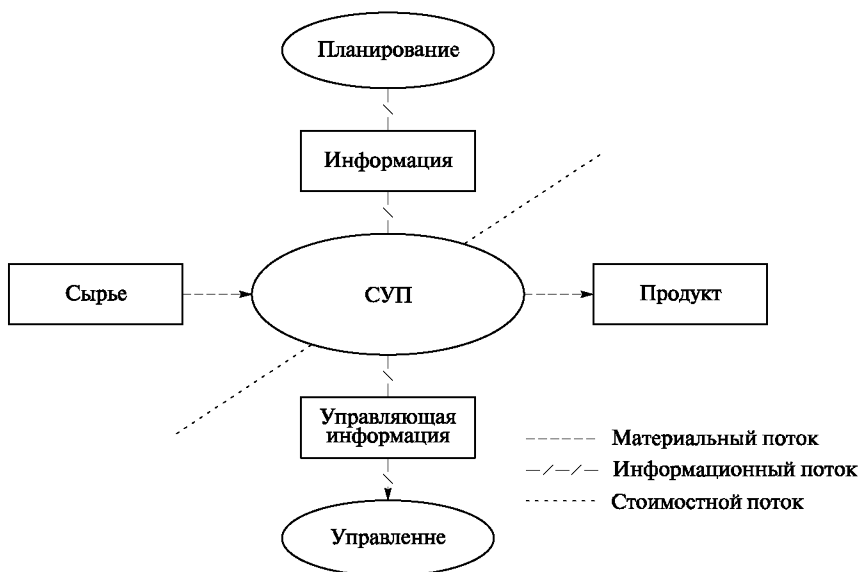


Рис. 1.6. Концептуальная модель функционирования СУП

- стоимостные потоки (финансовые аспекты).

Обычно с каждым типом потока связывают отдельные группы активностей и отдельные группы заинтересованных сторон.

Материальные потоки используют различные ресурсы, такие как сырье, денежные средства, человеческий труд с целью создания продукта. В рамках данного потока обычно можно выделить корневой процесс, соединяющий входы и выходы, который включает следующие типовые активности:

- приобретение (получение) сырья у поставщика;
- собственно изготовление;
- дистрибуция (поставка на рынок);
- продажа товара.

Информационный поток обеспечивает управление производством и планирование производства.

Стоимостной поток — это учет того, как увеличивается стоимость изделия на каждом этапе производственного процесса, фактически речь идет об учете расходов на каждой стадии производственного процесса.

Типовой подход к построению систем данного типа состоит в использовании иерархического подхода. Обычно выделяются три уровня, на которых осуществляется планирование и принятие решений:

- стратегический уровень планирования и принятия решений;
- тактический уровень планирования и принятия решений;
- оперативный уровень планирования и принятия решений.

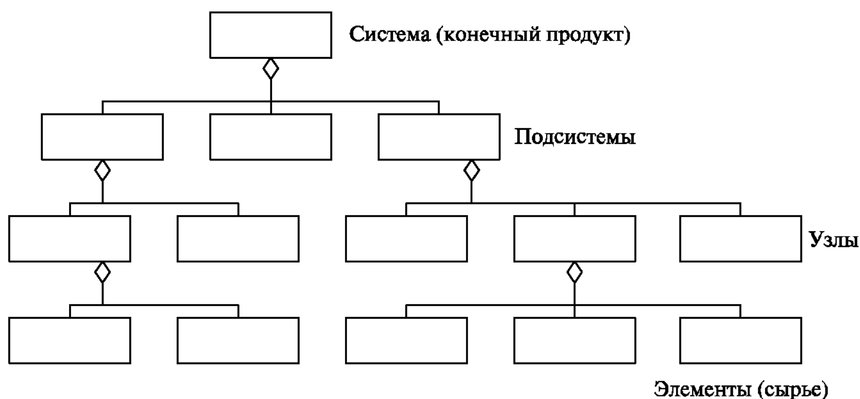


Рис. 1.7. Трехуровневая организация СУП

Часто конечный продукт является сложной системой и также имеет иерархическую структуру. Чаще всего используется четырехуровневая организация: элементы, узлы, подсистемы, система (рис. 1.7).

Системы управления доступом (СУД). К данному типу относятся системы, на которые возлагается решение задач, связанных с обеспечением доступа субъектов к объектам и ресурсам с использованием четко определенных политик и процедур.

Многие реальные системы реализуют данную функцию, хотя обычно она является одной из многих фупкций, реализуемых системой.

В качестве примеров систем управления доступом можно привести следующие системы:

- банкоматы;
- торговые автоматы;
- системы безопасности.

В основе фупкционирования систем управления доступом обычно используется хорошо известная эталонная модель управления доступом (Reference Monitor model), которая была разработана еще в 1960-х гг. и использовалась в качестве системы управления доступом как в мейнфреймах, так и мини-ЭВМ для обеспечения безопасности в режиме многопользовательского доступа к компьютерам. Песмотря на солидный возраст данная модель продолжает активно использоваться для построения систем безопасности, хотя основная сфера использования — это подсистемы безопасности, в которых она используется как вспомогательное приложение.

Обобщенная структура эталонной модели доступа (рис. 1.8) включает следующие основные элементы: субъекты, объекты, базу данных авторизаций (БДА), подсистему аудита безопасности (ПАБ) и движок.

Субъект представляет собой некоторую активную сущность, которая запрашивает доступ к ресурсу от имени определенного поль-

зователя. При входе в систему пользователь вводит свое имя и пароль. Пароль служит идентификатором, который известен только пользователю и системе. Система присваивает каждому пользователю упикальный идентификатор. Пользователи могут объединяться в группы. Каждый пользователь может участвовать в нескольких группах.

Объекты представляют собой пассивные хранилища информации, которые требуется защитить от несанкционированного доступа. Можно назвать следующие типовые объекты:

- файлы;
- директории;
- дисковые тома;
- сетевые объекты;
- почтовые ящики;
- очереди сообщений.

Система защищает объекты от несанкционированного доступа и предоставляет ряд механизмов, которые обеспечивают корректное совместное использование объектов несколькими пользователями.

База данных авторизаций хранит информацию о правах доступа к объектам. Чаще всего при проверке прав доступа к объекту со стороны субъекта используется идентификатор пользователя. Права доступа могут быть определены, например, в терминах того, кто является владельцем объекта, в терминах разрешения доступа членам группы.

Подсистема аудита безопасности отвечает за ведение журнала, в котором отмечаются успешные и неуспешные попытки входа в систему. После определенного числа безуспешных попыток входа в систему подсистема аудита может заблокировать дальнейшие попытки и выдать предупреждение администратору.

Движок представляет собой программный процессор, который реализует процедуру авторизации.

Можно сказать, что система управления доступом обеспечивает реализацию политики или политик доступа, основанных на правилах. К ней предъявляются следующие типовые требования:

- система должна обеспечивать надежную защиту объектов от несанкционированного доступа;

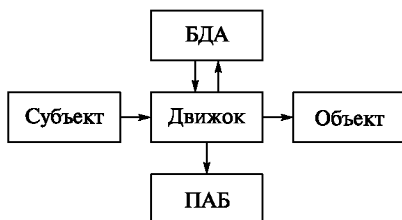


Рис.1.8. Обобщенная структура эталонной модели доступа

• система должна быть компактной и быстрой, поскольку все обращения к объектам проходят через данную систему.

Рассматриваемая эталонная модель может быть применена как для систем, работающих на одном компьютере, так и для распределенных систем.

Применительно к распределенным системам можно выделить три типовых подхода к реализации механизма управления доступом:

- прямое управление;
- мандатное управление;
- ролевое управление.

При использовании *прямого управления* все субъекты и объекты определены так, как это было описано выше применительно к эталонной модели. При использовании прямого управления субъекты-владельцы объекта могут разрешать или запрещать доступ к этим другим субъектам. Принципиально объекты и субъекты могут меняться ролями. Данный подход используется наиболее часто, хотя и не обеспечивает высоких показателей безопасности.

При использовании *мандатного управления* все объекты и субъекты относятся к определенному уровню. При использовании данного подхода субъекты не могут получить доступ к объектам, если им не разрешено работать на данном уровне.

Идея *ролевого управления* состоит в том, что определяется набор ролей, которые могут соответствовать, например, служебным обязанностям сотрудника или функциям, реализуемым подсистемой. Права доступа определяются, в частности, ролями, закрепленными за субъектом.

Ролевое управление доступом является наиболее гибким и простым с точки зрения администрирования.

В рамках конкретных систем возможно совместное использование рассмотренных подходов.

1.3. Архитектура и проектирование информационных систем

Архитектурное описание самым тесным образом связано с процессом проектирования ИС, причем в ряде определений термина «архитектура» на этот факт указывается в явном виде. Обычно выделяются пять различных подходов к проектированию, которые называют также стилями проектирования и, по существу, определяют группы методологий разработки ПО:

• календарный стиль — основанный на календарном планировании (Calendar-driven);

• стиль, основанный на управлении требованиями (Requirements-driven);

• стиль, в основу которого положен процесс разработки документации (Documentation-driven);

- стиль, основанный на управлении качеством (Quality-driven);
- архитектурный стиль (Architecture-driven).

Основой *календарного стиля* является график работ. Команда разработчиков выполняет работы поэтапно. Проектные решения принимаются из целей и задач конкретного этапа. Слабыми местами данного стиля являются то, что основные решения принимаются исходя из локальных целей, при этом мало внимания уделяется процессу разработки, разработке документации, созданию стабильных архитектур и внесению изменений. Все это приводит к высокой суммарной стоимости владением в долгосрочном плане. Данный стиль считается морально устаревшим, однако многие организации продолжают его использовать.

Стиль, основанный на управлении требованиями, предполагает, что основное внимание уделяется функциональным характеристикам системы, при этом часто недостаточно внимания уделяется нефункциональным характеристикам, таким как масштабируемость, портабельность, поддерживаемость и другим, определенным в ISO 9126. Проектные решения принимаются преимущественно исходя из локальных целей, связанных с реализацией тех или иных функций. Данный подход достаточно эффективен в случае, если требования определены и не изменяются в процессе проектирования. Основные недостатки данного подхода следующие: недостаточное внимание уделяется требованиям стандарта ISO 9126 (управление качеством); разрабатываемые архитектуры, как правило, не являются стабильными, так как каждая из реализуемых функций отображается на один или несколько функциональных компонентов. Это обстоятельство усложняет добавление к системе новых требований. Данный подход позволяет успешно отслеживать требования в плане реализации требуемой функциональности, однако использование его для долгосрочных проектов является неэффективным.

Стиль, в основу которого положен процесс разработки документации, до настоящего времени продолжает использоваться в правительственных организациях и крупных компаниях. Данный стиль может рассматриваться как вырожденный вариант стиля, основанного на управлении качеством, и ориентирован на разработку документации. В качестве основных недостатков данного подхода выделяются следующие: неоправданно много времени и сил затрачивается на разработку документации в ущерб качеству разрабатываемого кода. При этом создаваемая документация часто не используется ни пользователем, ни заказчиком.

Стиль, основанный на управлении качеством, предполагает самое широкое использование различных мер для отслеживания критичных с точки зрения функционирования параметров. Например, требуется получить время реакции системы на запрос менее одной секунды или обеспечить среднее время между отказами не менее 10 лет. В этом случае данные параметры отслеживаются на всех этапах

проектирования систем и часто это делается в ущерб другим характеристикам, таким как масштабируемость, простота сопровождения и т.п. Типовыми проблемами, возникающими при использовании данного стиля, являются следующие: часто оптимизируются не те параметры, которые должны быть в действительности, когда появляются новые требования, бывает очень трудно изменить функциональность системы. Созданные таким образом системы обычно не отличаются высоким качеством архитектурных решений. В целом данный стиль считается консервативным. Его использование может быть оправдано в случае, когда необходимо создать системы с экстремальными характеристиками.

Архитектурный стиль представляет собой наиболее зрелый подход к проектированию. В рамках данного стиля во главу угла ставится создание фреймворков, которые могут быть легко адаптированы ко всем потенциальным требованиям всех потенциальных заказчиком. Отличительная особенность данного стиля состоит в том, что задача проектирования разбивается на две отдельные подзадачи: создание многократно используемого фреймворка и создание конкретного приложения (системы) на его основе. При этом эти две задачи могут решать разные специалисты. Основная цель создания данного стиля — это устранение недостатков стиля, основанного на управлении требованиями. Использование архитектурного стиля позволяет реализовать инкрементное и итеративное проектирование, т.е. оперативно изменять существующую и добавлять новую функциональность.

В процессе проектирования важное значение приобретают атрибуты качества ИС.

Понятие качества ИС соответствует понятию о том, что система успешно справляется со всеми возлагаемыми на нее задачами, имеет хорошие показатели надежности и приемлемую стоимость, удобна в эксплуатации и обслуживании, легко сочетается с другими системами и в случае необходимости может быть модифицирована.

Разные группы пользователей имеют различные точки зрения на характеристики качества ИС. Например, если задать вопрос о том, какой должна быть хорошая ИС, то от пользователя можно получить следующие варианты ответов:

- система имеет хорошую производительность;
- система имеет широкие функциональные возможности;
- система удобна в эксплуатации;
- система надежна.

Менеджер даст скорее всего другие варианты ответов:

- стоимость системы не должна быть изначально очень высокой;
- система не должна быть очень дорогой в эксплуатации;
- система не должна морально устаревать в течение возможно более длительного периода времени и в случае необходимости может быть легко модифицирована.

Для системного администратора наиболее важными могут оказаться такие характеристики системы, как

- надежность и стабильность работы;
- простота администрирования;
- наличие хорошей эксплуатационной документации;
- хорошая поддержка изготовителем.

Другие заинтересованные лица могут иметь свои точки зрения на то, какой должна быть качественная система.

Поскольку в современных ИС ключевой компонентой является программная компонента, а пользователи, работающие с системой, в большинстве случаев взаимодействуют непосредственно с программной компонентой, поэтому показатели качества информационных и программных систем в значительной степени совпадают.

Для того чтобы построить правильную и надежную архитектуру и грамотно спроектировать интеграцию программных систем, необходимо четко следовать современным стандартам в этих областях. Без этого велика вероятность создать архитектуру, которая неспособна развиваться и удовлетворять растущие потребности пользователей ИТ. В качестве законодателей стандартов в этой области выступают такие международные организации, как SEI (Software Engineering Institute), WWW (консорциум World Wide Web), OMG (Object Management Group), организация разработчиков Java — JCP (Java Community Process), IEEE (Institute of Electrical and Electronics Engineers) и др.

Качество программного обеспечения определяется стандартом ISO 9126 [42] как вся совокупность его характеристик, относящихся к возможности удовлетворять высказанные или подразумеваемые потребности всех заинтересованных лиц.

Различаются понятия внутреннего качества, связанного с характеристиками программного обеспечения (ПО) самого по себе, без учета его поведения; внешнего качества, характеризующего ПО с точки зрения его поведения; и качества ПО при использовании в различных контекстах — того качества, которое ощущается пользователями при конкретных сценариях работы ПО. Для всех этих аспектов качества введены метрики, позволяющие оценить их. Кроме того, для создания добротного ПО существенное значение имеет качество технологических процессов его разработки.

ISO 9126 — это международный стандарт, определяющий оценочные характеристики качества программного обеспечения. Российский аналог стандарта ГОСТ 28195. Стандарт разделяется на четыре части, описывающие следующие вопросы: модель качества, внешние метрики качества, внутренние метрики качества, метрики качества в использовании.

Вторая и третья части стандарта ISO 9126-2,3 посвящены формализации соответственно внешних и внутренних метрик характеристик качества сложных программных систем. В ней изложены содер-

жание и общие рекомендации по использованию соответствующих метрик и взаимосвязей между типами метрик.

Четвертая часть стандарта ISO 9126-4 предназначена для покупателей, поставщиков, разработчиков, сопровождающих, пользователей и менеджеров качества ПС. В ней повторена концепция трех типов метрик, а также аннотированы рекомендуемые виды измерений характеристик.

Модель качества, установленная в первой части стандарта ISO 9126-1, классифицирует качество ПО в шести структурных наборах характеристик:

- функциональность;
- надежность;
- производительность (эффективность);
- удобство использования (практичность);
- удобство сопровождения;
- переносимость.

Перечисленные характеристики, в свою очередь, детализированы подхарактеристиками (субхарактеристиками) (рис. 1.9). Ниже приведены определения этих характеристик и атрибутов по стандарту ISO 9126:2001.

Функциональность (functionality) определяется как способность ПО в определенных условиях решать задачи, нужные пользователям.

Для данной характеристики выделяются следующие субхарактеристики:

- функциональная пригодность;
- точность;
- способность к взаимодействию;
- защищенность;
- соответствие стандартам и правилам.

Функциональная пригодность (suitability) определяется как способность решать нужный набор задач.

Точность (accuracy) — определяется как способность выдавать нужные результаты.

Способность к взаимодействию (interoperability) — способность взаимодействовать с нужным набором других систем.

Соответствие стандартам и правилам (compliance) — соответствие ПО имеющимся отраслевым стандартам, нормативным и законодательным актам, другим регулирующим нормам.

Защищенность (security) — способность предотвращать неавторизированный, т.е. без указания лица, пытающегося его осуществить, и неразрешенный доступ к данным и программам.

Надежность (reliability) — способность ПО поддерживать определенную работоспособность в заданных условиях.

Для данной характеристики выделяются следующие субхарактеристики:

- зрелость (завершенность);

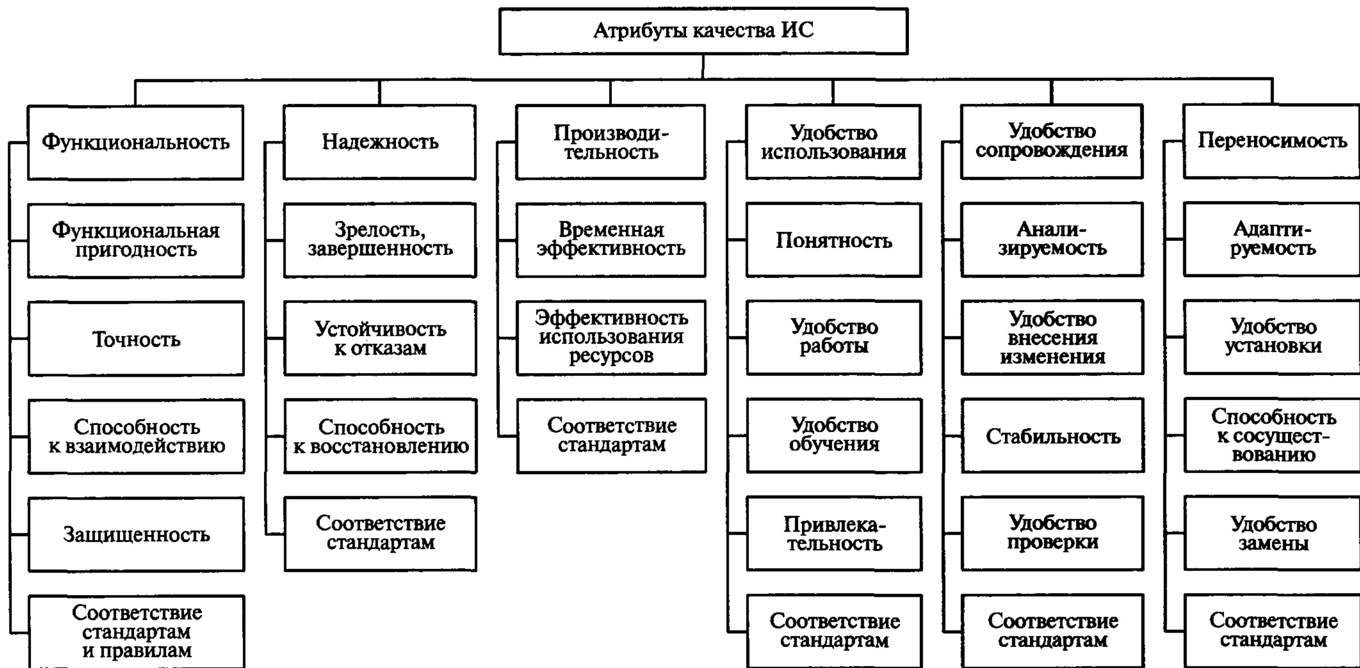


Рис.1.9. Характеристики качества ИС

• устойчивость к отказам;
• способность к восстановлению;
• соответствие стандартам надежности (reliability compliance).

Зрелость, завершенность (maturity) — величина, обратная частоте отказов ПО. Обычно измеряется средним временем работы без сбоев и величиной, обратной вероятности возникновения отказа за данный период времени.

Устойчивость к отказам (fault tolerance) — способность поддерживать заданный уровень работоспособности при отказах и нарушениях правил взаимодействия с окружением.

Способность к восстановлению (recoverability) определяется как способность восстанавливать определенный уровень работоспособности и целостность данных после отказа, необходимые для этого время и ресурсы.

Производительность (efficiency), или **эффективность**, — способность ПО при заданных условиях обеспечивать необходимую работоспособность по отношению к выделяемым для этого ресурсам. Можно определить ее и как отношение получаемых с помощью ПО результатов к затрачиваемым на это ресурсам всех типов.

Для данной характеристики выделяются следующие субхарактеристики:

• временная эффективность;
• эффективность использования ресурсов;
• соответствие стандартам производительности (efficiency compliance).

Временная эффективность (time behaviour) — способность ПО выдавать ожидаемые результаты, а также обеспечивать передачу необходимого объема данных за отведенное время.

Эффективность использования ресурсов (resource utilisation) — способность решать нужные задачи с использованием определенных объемов ресурсов определенных видов. Имеются в виду такие ресурсы, как оперативная и долговременная память, сетевые соединения, устройства ввода и вывода и пр.

Удобство использования (usability), или **практичность**, определяется как способность ПО быть удобным в обучении и использовании, а также привлекательным для пользователей.

Для данной характеристики выделяются следующие субхарактеристики:

• понятность;
• удобство работы;
• удобство обучения;
• привлекательность;
• соответствие стандартам удобства использования (usability compliance).

Понятность (understandability) — это показатель, обратный усилиям, которые затрачиваются пользователями на восприятие основ-

ных понятий ПО и осознание их применимости для решения своих задач.

Удобство работы (operability) — это показатель, обратный усилиям, предпринимаемым пользователями для решения своих задач с помощью ПО.

Удобство обучения (learnability) — показатель, обратный усилиям, затрачиваемым пользователями на обучение работе с ПО.

Привлекательность (attractiveness) — это способность ПО быть привлекательным для пользователей. Этот атрибут добавлен в 2001 г.

Удобство сопровождения (maintainability) определяется как удобство проведения всех видов деятельности, связанных с сопровождением программ.

Для данной характеристики выделяются следующие субхарактеристики:

- анализируемость;
- удобство внесения изменений;
- стабильность;
- удобство проверки;
- соответствие стандартам удобства сопровождения (maintainability compliance).

Анализируемость (analyzability), или удобство проведения анализа, определяется как удобство проведения анализа ошибок, дефектов и недостатков, а также удобство анализа необходимости изменений и их возможных последствий.

Удобство внесения изменений (changeability) — показатель, обратный трудозатратам на выполнение необходимых изменений.

Стабильность (stability) — показатель, обратный риску возникновения неожиданных эффектов при внесении необходимых изменений.

Удобство проверки (testability) — показатель, обратный трудозатратам на проведение тестирования и других видов проверки того, что внесенные изменения привели к нужным результатам.

Переносимость (portability) определяется как способность ПО сохранять работоспособность при переносе из одного окружения в другое, включая организационные, аппаратные и программные аспекты окружения.

Иногда эта характеристика называется в русскоязычной литературе мобильностью. Однако термин «мобильность» стоит зарезервировать для перевода «mobility» — способности ПО и компьютерной системы в целом сохранять работоспособность при ее физическом перемещении в пространстве.

Для данной характеристики выделяются следующие субхарактеристики:

- адаптируемость;
- удобство установки;

- способность к сосуществованию;
- удобство замены;
- соответствие стандартам переносимости (portability compliance).

Адаптируемость (adaptability) — способность ПО приспосабливаться к различным окружениям без проведения для этого действий помимо заранее предусмотренных.

Удобство установки (installability) — способность ПО быть установленным или развернутым в определенном окружении.

Способность к сосуществованию (coexistence) — способность ПО сосуществовать с другими программами в общем окружении, деля с ними ресурсы.

Удобство замены (replaceability) другого ПО данным определяется как возможность применения данного ПО вместо других программных систем для решения тех же задач в определенном окружении.

Перечисленные атрибуты относятся к внутреннему и внешнему качеству ПО согласно ISO 9126. Для описания качества ПО при использовании стандарт ISO 9126-4 предлагает другой, более узкий набор характеристик:

- эффективность;
- продуктивность;
- безопасность;
- удовлетворение пользователей.

Эффективность (effectiveness) — способность ПО предоставлять пользователям возможность решать их задачи с необходимой точностью при использовании в заданном контексте.

Продуктивность (productivity) — способность ПО предоставлять пользователям определенные результаты в рамках ожидаемых затрат ресурсов.

Безопасность (safety) — способность ПО обеспечивать необходимо низкий уровень риска нанесения ущерба жизни и здоровью людей, бизнесу, собственности или окружающей среде.

Удовлетворение пользователей (satisfaction) — способность ПО приносить удовлетворение пользователям при использовании в заданном контексте.

1.4. Эволюция платформенных архитектур информационных систем

Развитие платформенных архитектур ИС происходило по трем направленям: автономные, централизованные и распределенные архитектуры. Развитие этих направлений связано с необходимостью обеспечения интеграции и построения единого информационного пространства.

Подробно рассмотрим особенности каждого направления развития архитектуры ИС [5].

Автономными (standalone) *аритектурами* могут быть сервисные программы, системные утилиты, текстовые и графические редакторы, компиляторы, достаточно простые корпоративные программы. Развитая корпоративная информационная система, как правило, не может состоять из отдельных, не связанных между собой компонентов.

Централизованная архитектура вычислительных систем была распространена в 1970—1980-х гг. и реализовывалась на базе мейнфреймов (например, IBM-360/370 или их отечественных аналогов серии ЕС ЭВМ) либо на базе мини-ЭВМ (например, PDP-11 или их отечественного аналога СМ-4). Характерная особенность такой архитектуры — полная «неинтеллектуальность» терминалов. Их работой управляет хост-ЭВМ.

Достоинства такой архитектуры:

- пользователи совместно используют дорогие ресурсы ЭВМ и дорогие периферийные устройства;
- централизация ресурсов и оборудования облегчает обслуживание и эксплуатацию вычислительной системы;
- отсутствует необходимость администрирования рабочих мест пользователей;

Главным недостатком для пользователя является то, что он полностью зависит от администратора хост-ЭВМ. Пользователь не может настроить рабочую среду под свои потребности — все используемое программное обеспечение является коллективным.

Использование такой архитектуры является оправданным, если хост-ЭВМ очень дорогая, например суперЭВМ.

Классическое представление *централизованной архитектуры* показано на рис. 1.10.

Центральная ЭВМ должна иметь большую память и высокую производительность, чтобы обеспечивать комфортную работу большого числа пользователей.

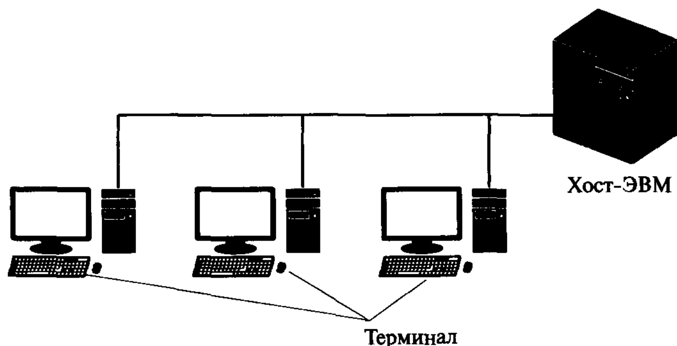


Рис. 1.10. Классическое представление централизованной архитектуры

Все программы выполняются на хост-ЭВМ, а терминалы являются лишь устройствами ввода-вывода и, таким образом, в минимальной степени поддерживают интерфейс пользователя.

Распределенные архитектуры имеют более богатую историю, что связано с бурным развитием технических и программных компонент.

Существует шесть основных характеристик архитектур распределенных систем.

- совместное использование ресурсов: распределенные системы допускают совместное использование как аппаратных (жестких дисков, принтеров), так и программных (файлов, компиляторов) ресурсов;
- открытость — возможность расширения системы путем добавления новых ресурсов;
- параллельность: в распределенных системах несколько процессов могут одновременно выполняться на разных компьютерах в сети. Эти процессы могут взаимодействовать во время их выполнения;
- масштабируемость — возможность добавления новых свойств и методов;
- отказоустойчивость: наличие нескольких компьютеров позволяет дублирование информации и устойчивость к некоторым аппаратным и программным ошибкам. Распределенные системы в случае ошибки могут поддерживать частичную функциональность. Полный сбой в работе системы происходит только при сетевых ошибках;
- прозрачность: пользователям предоставляется полный доступ к ресурсам в системе, в то же время от них скрыта информация о распределении ресурсов по системе.

Распределенные системы обладают и рядом недостатков:

- сложность: намного труднее понять и оценить свойства распределенных систем в целом, их сложнее проектировать, тестировать и обслуживать. Также производительность системы зависит от скорости работы сети, а не отдельных процессоров. Перераспределение ресурсов может существенно изменить скорость работы системы;
- безопасность: обычно доступ к системе можно получить с нескольких разных машин, сообщения в сети могут просматриваться и перехватываться. Поэтому в распределенной системе намного труднее поддерживать безопасность;
- управляемость: система может состоять из разнотипных компьютеров, на которых могут быть установлены различные версии операционных систем. Ошибки на одной машине могут распространиться непредсказуемым образом на другие машины;
- непредсказуемость: реакция распределенных систем на некоторые события непредсказуема и зависит от полной загрузки системы, ее организации и сетевой нагрузки. Так как эти параметры могут постоянно изменяться, поэтому время ответа на запрос может существенно отличаться от времени.

Эти недостатки учитывают при проектировании распределенных систем. Однако при этом возникает ряд проблем, которые надо учитывать разработчикам.

Ресурсы в распределенных системах располагаются на разных компьютерах, поэтому систему имен ресурсов следует продумать так, чтобы пользователи могли без труда открывать необходимые им ресурсы, т.е. *идентифицировать ресурсы* и ссылаться на них. Примером может служить система URL(унифицированный указатель ресурсов), которая определяет имена Web-страниц.

Универсальная работоспособность Internet и эффективная реализация протоколов TCP/IP в Internet для большинства распределенных систем служат примером наиболее эффективного способа организации взаимодействия (*коммуникация*) между компьютерами. Однако в некоторых случаях, когда требуется особая производительность или надежность, возможно использование специализированных средств.

Качество системного сервиса отражает производительность, работоспособность и надежность. На качество сервиса влияет ряд факторов: распределение процессов, ресурсов, аппаратные средства и возможности адаптации системы.

Архитектура программного обеспечения описывает распределение системных функций по компонентам системы, а также распределение этих компонентов по процессорам. Если необходимо поддерживать высокое качество системного сервиса, выбор правильной архитектуры является решающим фактором.

Задача разработчиков распределенных систем — спроектировать программное и аппаратное обеспечение так, чтобы предоставить все необходимые характеристики распределенной системы. А для этого требуется знать преимущества и недостатки различных архитектур распределенных систем.

Выделяют следующие виды архитектур распределенных информационных систем:

- архитектура «файл-сервер»;
- архитектура «клиент-сервер»;
- архитектура Web-приложений.

Рассмотрим каждую из этих архитектур.

Архитектура «файл-сервер». Файл-серверные приложения схожи по своей структуре с локальными приложениями и используют сетевой ресурс для хранения программ и данных.

Классическое представление информационной системы в архитектуре «файл-сервер» представлено на рис. 1.11.

Организация информационных систем на основе использования выделенных файл-серверов все еще является распространенной в связи с наличием большого количества персональных компьютеров разного уровня развитости и сравнительной дешевизны связывания PC в локальные сети. Основным достоинством данной архитектуры явля-

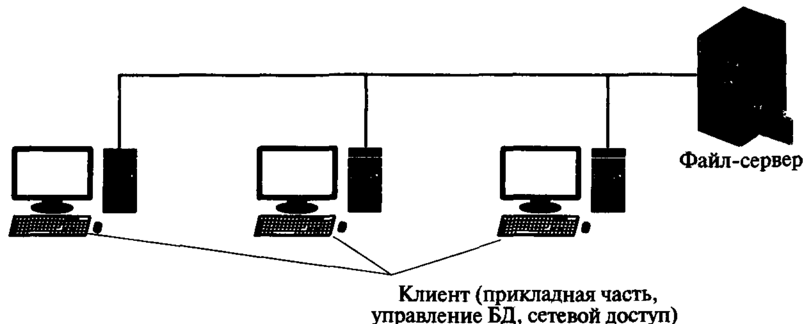


Рис. 1.11. Классическое представление архитектуры «файл-сервер»

ется простота организации. Проектировщики и разработчики информационной системы находятся в привычных и комфортных условиях IBM PC в среде MS-DOS, Windows или какого-либо облегченного варианта Windows Server. Имеются удобные и развитые средства разработки графического пользовательского интерфейса, простые в использовании средства разработки систем баз данных и/или СУБД.

Достоинства такой архитектуры:

- многопользовательский режим работы с данными;
- удобство централизованного управления доступом;
- низкая стоимость разработки;
- высокая скорость разработки;
- невысокая стоимость обновления и изменения ПО.

Недостатки:

- проблемы многопользовательской работы с данными — последовательный доступ, отсутствие гарантии целостности;
- низкая производительность (зависит от производительности сети, сервера, клиента);
- плохая возможность подключения новых клиентов;
- ненадежность системы.

Простое, работающее с небольшими объемами информации и рассчитанное на применение в однопользовательском режиме, файл-серверное приложение можно спроектировать, разработать и отладить очень быстро. Очень часто для небольшой компании для ведения, например кадрового учета, достаточно иметь изолированную систему, работающую на отдельно стоящем PC. Однако в уже более сложных случаях (например, при организации информационной системы поддержки проекта, выполняемого группой) файл-серверные архитектуры становятся недостаточными.

Архитектура «клиент-сервер» (Client-server). Представляет собой вычислительную или сетевую архитектуру, в которой задания или сетевая нагрузка распределены между поставщиками услуг (сервисов), называемых серверами, и заказчиками услуг, называемых клиентами.

Нередко клиенты и серверы взаимодействуют через компьютерную сеть и могут быть как различными физическими устройствами, так и программным обеспечением.

Первоначально системы такого уровня базировались на классической двухуровневой клиент-серверной архитектуре (Two-tier architecture). Под клиент-серверным приложением в этом случае понимается информационная система, основанная на использовании серверов баз данных.

Схематически такую архитектуру можно представить, как показано на рис. 1.12.

На стороне клиента выполняется код приложения, в который обязательно входят компоненты, поддерживающие интерфейс с конечным пользователем и производящие отчеты, а также другие специфичные для приложения функции.

Клиентская часть приложения взаимодействует с клиентской частью программного обеспечения управления базами данных, которая, фактически является индивидуальным представителем СУБД для приложения.

Заметим, что интерфейс между клиентской частью приложения и клиентской частью сервера баз данных, как правило, основан на использовании языка SQL. Поэтому такие функции, как например предварительная обработка форм, предназначенных для запросов к базе данных, или формирование результирующих отчетов, выполняются в коде приложения.

Наконец, клиентская часть сервера баз данных, используя средства сетевого доступа, обращается к серверу баз данных, передавая ему текст оператора языка SQL.

Посмотрим теперь, что же происходит на стороне *сервера баз данных*. В продуктах практически всех компаний сервер получает от клиента текст оператора на языке SQL. Сервер производит компиляцию полученного оператора. Далее (если компиляция завершилась успешно) происходит выполнение оператора.

Рис. 1.12. Классическое представление архитектуры «клиент-сервер»

Разработчики и пользователи информационных систем, основанных на архитектуре «клиент-сервер», часто бывают неудовлетворены постоянно существующими сетевыми накладными расходами, возникающими из-за потребности обращаться от клиента к серверу с каждым очередным запросом. На практике распространена ситуация, когда для эффективной работы отдельной клиентской составляющей информационной системы в действительности требуется только небольшая часть общей базы данных. Это приводит к идее поддержки локального кэша общей базы данных на стороне каждого клиента.

Фактически концепция локального кэширования базы данных является частным случаем концепции реплицированных баз данных. Как и в общем случае, для поддержки локального кэша базы данных программное обеспечение рабочих станций должно содержать компонент управления базами данных — упрощенный вариант сервера баз данных, который, например, может не обеспечивать многопользовательский режим доступа.

Отдельной проблемой является обеспечение согласованности (когерентности) кэшей и общей базы данных. Здесь возможны различные решения — от автоматической поддержки согласованности за счет средств базового программного обеспечения управления базами данных до полного перекладывания этой задачи на прикладной уровень.

Преимущества данной архитектуры:

• возможность, в большинстве случаев, распределить функции вычислительной системы между несколькими независимыми компьютерами в сети;

• все данные хранятся на сервере, который, как правило, защищен гораздо лучше большинства клиентов, а также на сервере проще обеспечить контроль полномочий, чтобы разрешать доступ к данным только клиентам с соответствующими правами доступа;

• поддержка многопользовательской работы;

• гарантия целостности данных.

Недостатки:

• неработоспособность сервера может сделать неработоспособной всю вычислительную сеть;

• администрирование данной системы требует квалифицированного профессионала;

• высокая стоимость оборудования;

• бизнес-логика приложений осталась в клиентском ПО.

При проектировании информационной системы, основанной на архитектуре «клиент-сервер», большее внимание следует обращать на грамотность общих решений. Технические средства пилотной версии могут быть минимальными (например, в качестве аппаратной основы сервера баз данных может использоваться одна из рабочих станций). После создания пилотной версии нужно провести допол-

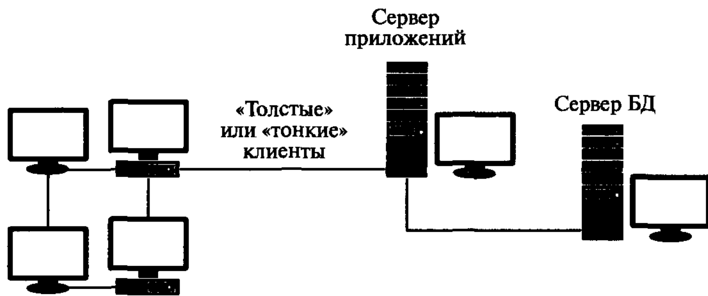


Рис. 1.13. Многозвенная архитектура

нительную исследовательскую работу, чтобы выяснить узкие места системы. Только после этого необходимо принимать решение о выборе аппаратуры сервера, которая будет использоваться на практике.

Увеличение масштабов информационной системы не порождает принципиальных проблем. Обычным решением является замена аппаратуры сервера (и, может быть, аппаратуры рабочих станций, если требуется переход к локальному кэшированию баз данных). В любом случае практически не затрагивается прикладная часть информационной системы.

Данный вид архитектуры также называют архитектурой с «толстым» клиентом.

В случае большого числа пользователей возникают проблемы своевременной и синхронной замены версий клиентских приложений на рабочих станциях. Такие проблемы решаются в рамках многозвенной архитектуры (рис. 1.13). Часть общих приложений переносится на специально выделенный сервер приложений. Тем самым понижаются требования к ресурсам рабочих станций, которые будут называться «тонкими» клиентами. Данный способ организации вычислительного процесса является разновидностью архитектуры «клиент-сервер».

Использование многозвенной архитектуры может быть рекомендовано также тогда, когда некоторая программа требует для своей работы много ресурсов. В этом случае может оказаться дешевле построить сеть с одним очень мощным сервером, чем использовать несколько мощных клиентных рабочих станций. Особенно это имеет значение, если данной программой пользуются не постоянно, а время от времени. На рис. 1.14 приведена архитектура многозвенного приложения.

Разумное сочетание производительности сервера приложений и производительности рабочих станций позволят построить сеть, более дешевую при установке и эксплуатации.

Архитектура Web-приложений (*архитектура Web-сервисов*). Эта архитектура широко применяется в настоящее время. Web-сервис — приложение, доступное через Internet и предоставляющее не-

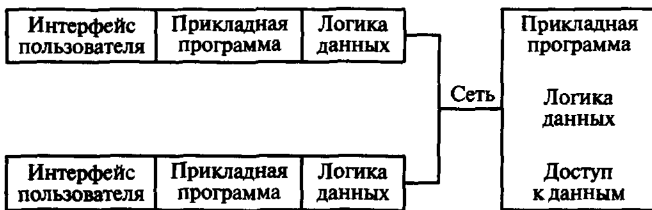


Рис. 1.14. Архитектура многозвенного приложения

которые услуги, форма которых не зависит от поставщика (так как используется универсальный формат данных — XML) и платформы функционирования. В данное время существуют три различные технологии, поддерживающие концепцию распределенных объектных систем: EJB, DCOM и CORBA.

В основе Web-сервисов лежат открытые стандарты и протоколы (SOAP, UDDI и WSDL):

- SOAP (Simple Object Access Protocol), разработанный консорциумом W3C, определяет формат запросов к Web-сервисам. Сообщения между Web-сервисом и его пользователем пакуются в так называемые SOAP-конверты (SOAP envelopes, иногда их еще называют XML-конвертами). Само сообщение может содержать либо запрос на осуществление какого-либо действия, либо ответ — результат выполнения этого действия;
- WSDL (Web Service Description Language). Интерфейс Web-сервиса описывается в WSDL-документах (а WSDL — это подмножество XML). Перед развертыванием службы разработчик составляет ее описание на языке WSDL, указывает адрес Web-сервиса, поддерживаемые протоколы, перечень допустимых операций, форматы запросов и ответов;
- UDDI (Universal Description, Discovery and Integration) — протокол поиска Web-сервисов в Internet (http://www.uddi.org/). Представляет собой бизнес-реестр, в котором провайдеры Web-сервисов регистрируют службы, а разработчики находят необходимые сервисы для включения в свои приложения.

Основная идея, лежавшая в разработке *технологии EJB* (Enterprise JavaBeans) — создать такую инфраструктуру для компонентов, чтобы они могли бы легко «вставляться» («plug in») и удаляться из серверов, тем самым повышая или снижая функциональность сервера. Технология Enterprise JavaBeans похожа на технологию JavaBeans в том смысле, что она использует ту же самую идею (а именно, создание нового компонента из уже существующих, готовых и настраиваемых, аналогично RAD-системам), но во всем остальном Enterprise JavaBeans — совершенно иная технология.

Опубликованная в марте 1998 г. EJB-спецификация версии 1.0 определяет следующие цели.

1. Облегчить разработчикам создание приложений, избавив их от необходимости реализовывать с нуля такие сервисы, как транзакции (transactions), нити (threads), загрузка (load balancing) и другие. Разработчики могут сконцентрироваться на описании логики своих приложений, оставляя заботы о хранении, передаче и безопасности данных на EJB-систему. При этом все равно имеется возможность самому контролировать и описывать порученные системе процессы.

2. Описать основные структуры EJB-системы, описав при этом интерфейсы взаимодействия (contracts) между ее компонентами.

3. EJB преследует цель стать стандартом для разработки клиент/сервер приложений на Java. Таким же образом, как исходные JavaBeans (Delphi, или другие) компоненты от различных производителей можно было составлять вместе с помощью соответствующих RAD-систем, получая в результате работоспособные клиенты, таким же образом серверные компоненты EJB от различных производителей также могут быть использованы вместе. EJB-компоненты, будучи Java-классами, должны, без сомнения, работать на любом EJB-совместимом сервере даже без перекомпиляции, что практически нереально для других систем.

4. EJB совместима с Java API, может взаимодействовать с другими (не обязательно Java) приложениями, а также совместима с CORBA.

Разработчику, однако, не нужно самому реализовывать EJB-объект. Этот класс создается специальным кодогенератором, поставляемым вместе в EJB-контейнером. Как уже было сказано, EJB-объект (созданный с помощью сервисов контейнера) и EJB-компонента (созданная разработчиком) реализуют один и тот же интерфейс. В результате, когда приложение-клиент хочет вызвать метод у EJB-компоненты, то сначала вызывается аналогичный (по имени) метод у EJB-объекта, что находится на стороне клиента, а тот, в свою очередь, связывается с удаленной EJB-компонентой и вызывает у нее этот метод (с теми же аргументами).

Рассмотрим два различных типа «бинов».

Первый тип *Session bean* представляет собой EJB-компоненту, связанную с одним клиентом. «Бины» этого типа, как правило, имеют ограниченный срок жизни (хотя это и не обязательно) и редко участвуют в транзакциях. В частности, они обычно не восстанавливаются после сбоя сервера. В качестве примера session bean можно взять «бин», который живет в Web-сервере и динамически создает HTML-страницы клиенту, при этом следя за тем, какая именно страница загружена у клиента. Когда же пользователь покидает Web-узел, или по истечении некоторого времени, session bean уничтожается. Несмотря на то, что в процессе своей работы session bean мог сохранять некоторую информацию в базе данных, его предназначение заключается все-таки не в отображении состояния или в работе с «вечными объектами», а просто в выполнении некоторых функций на стороне сервера от имени одного клиента.

Второй тип *Entity bean*, наоборот, представляет собой компоненту, работающую с постоянной (persistent) информацией, хранящейся, например в базе данных. Entity beans ассоциируются с элементами баз данных и могут быть доступны одновременно нескольким пользователям. Так как информация в базе данных является постоянной, то и entity beans живут постоянно, «выживая», тем самым, после сбоев сервера (когда сервер восстанавливается после сбоя, он может восстановить «бин» из базы данных). Например, entity bean может представлять собой строку какой-нибудь таблицы из базы данных или даже результат операции SELECT. В объектно-ориентированных базах данных entity bean может представлять собой отдельный объект со всеми его атрибутами и связями.

Достоинства EJB:

- быстрое и простое создание;
- Java-оптимизация;
- кроссплатформенность;
- динамическая загрузка компонент-переходников;
- возможность передачи объектов по значению;
- встроенная безопасность.

Недостатки EJB:

- поддержка только одного языка — Java;
- трудность интегрирования с существующими приложениями;
- плохая масштабируемость;
- производительность;
- отсутствие международной стандартизации.

Благодаря своей легко используемой Java-модели, EJB является относительно простым и быстрым способом создания распределенных систем. EJB — хороший выбор для создания RAD-компонент и корпоративных приложений на языке Java. Технология EJB по своим возможностям не уступает DCOM. Тем самым, роль RMI в создании больших, масштабируемых промышленных систем снижается.

Технология DCOM (Distributed Component Object Model) — программная архитектура, разработанная компанией Microsoft для распределения приложений между несколькими компьютерами в сети. Программный компонент на одной из машин может использовать DCOM для передачи сообщения (его называют удаленным вызовом процедуры) к компоненту на другой машине. DCOM автоматически устанавливает соединение, передает сообщение и возвращает ответ удаленного компонента.

Для того чтобы различные фрагменты сложного приложения могли работать вместе через Internet, необходимо обеспечить между ними надежные и защищенные соединения, а также создать специальную систему, которая направляет программный трафик.

Для решения этой задачи компания Microsoft создала распределенную компонентную объектную модель Distributed Component Object Model (DCOM), которая встраивается в операционные системы Windows NT 4.0 и Windows 98 и выше.

Преимуществом DCOM является, по мнению Карен Буше, аналитика The Standish Group, значительная простота использования. Если программисты пишут свои Windows-приложения с помощью ActiveX (предлагаемого Microsoft способа организации программных компонентов), то операционная система будет автоматически устанавливать необходимые соединения и перенаправлять трафик между компонентами, независимо от того, размещаются ли компоненты на той же машине или нет.

Способность DCOM связывать компоненты позволила Microsoft наделить Windows рядом важных дополнительных возможностей, в частности, реализовать сервер Microsoft Transaction Server, отвечающий за выполнения транзакций баз данных через Internet. Новая же версия COM+ еще больше упростит программирование распределенных приложений, в частности, благодаря таким компонентам, как базы данных, размещаемые в оперативной памяти.

Однако у DCOM есть и ряд недостатков. «На самом деле это решение до сих пор ориентировано исключительно на системы Microsoft», — считает Буше. Изначально DCOM создавалась под Windows. Хорошо известно, что Microsoft заключила соглашение с компанией Software AG, предмет которого — перенос DCOM на другие платформы. Впрочем, по мнению Буше, значение этой работы достаточно ограниченно, поскольку Microsoft уже успела внести ряд существенных изменений в Windows-версию DCOM.

В числе недостатков и то, что архитектура предусматривает использование для поиска компонентов в сети, разработанной Microsoft сетевой службы каталогов Active Directory. Но эта служба каталогов появилась только в версии Windows 2000. В более ранних версиях DCOM должна использовать локальные списки компонентов, что совершенно неприемлемо для приложений большего масштаба, нежели рабочая группа, поскольку информация об изменении местонахождения компонента должна вручную заноситься в каждый работающий в сети компьютер.

Достоинства DCOM:

- независимость от языка;
- динамический/статический вызов;
- динамическое нахождение объектов;
- масштабируемость;
- открытый стандарт (контроль со стороны TOG);
- множественность Windows-программистов.

Недостатки DCOM:

- сложность реализации;
- зависимость от платформы;
- нет именования через URL;
- нет проверки безопасности на уровне выполнении ActiveX компонент;
- отсутствие альтернативных разработчиков.

DCOM является лишь частным решением проблемы распределенных объектных систем. Он хорошо подходит для Microsoft-ориентированных сред. Как только в системе возникает необходимость работать с архитектурой, отличной от Windows, DCOM перестает быть оптимальным решением проблемы. Конечно, вскоре это положение может измениться, так как Microsoft стремится перенести DCOM и на другие платформы. Например, фирмой Software AG была выпущена версия DCOM для Solaris UNIX и планируется выпуск версий и для других версий UNIX. Но все-таки, на сегодняшний день, DCOM хорош лишь в качестве решения для систем, ориентированных исключительно на продукты Microsoft. Большие нарекания вызывает также отсутствие безопасности при исполнении ActiveX компонент, что может привести к неприятным последствиям.

В конце 1980-х — начале 1990-х гг. многие ведущие фирмы-разработчики были заняты поиском технологий, которые принесли бы ощутимую пользу на все более изменчивом рынке компьютерных разработок. В качестве такой технологии была определена область распределенных компьютерных систем. Необходимо было разработать единообразную архитектуру, которая позволяла бы осуществлять повторное использование и интеграцию кода, что было особенно важно для разработчиков. Цена за повторное использование кода и интеграцию кода была высока, но никто из разработчиков в одиночку не мог воплотить в реальность мечту о широко используемом, языково-независимом стандарте, включающем в себя поддержку сложных многосвязных приложений. Поэтому в мае 1989 г. была сформирована OMG (Object Managment Group). Как уже отмечалось, сегодня OMG насчитывает более 700 членов (в OMG входят практически все крупнейшие производители ПО, за исключением Microsoft).

Задачей консорциума OMG является определение набора спецификаций, позволяющих строить интероперабельные информационные системы. Спецификация OMG — The Common Object Request Broker Architecture (CORBA) является индустриальным стандартом, описывающим высокоуровневые средства поддерживания взаимодействия объектов в распределенных гетерогенных средах.

Технология CORBA специфицирует инфраструктуру взаимодействия компонент (объектов) на представительском уровне и уровне приложений модели OSI. Она позволяет рассматривать все приложения в распределенной системе как объекты. Причем объекты могут одновременно играть роль и клиента, и сервера: роль клиента, если объект является инициатором вызова метода у другого объекта; роль сервера, если другой объект вызывает на нем какой-нибудь метод. Объекты-серверы обычно называют «реализацией объектов». Практика показывает, что большинство объектов одновременно исполняют роль и клиентов, и серверов, попеременно вызывая методы на других объектах и отвечая на вызовы извне. Использование CORBA позволит строить гораздо более гибкие системы, чем систе-

мы «клиент-сервер», основанные на двухуровневой и трехуровневой архитектурах.

Вот небольшой список достоинств и недостатков использования технологии CORBA.

Достоинства CORBA:

- платформенная независимость;
- языковая независимость;
- динамические вызовы;
- динамическое обнаружение объектов;
- масштабируемость;
- CORBA-сервисы;
- широкая индустриальная поддержка.

Недостатки CORBA:

- нет передачи параметров по значению;
- отсутствует динамическая загрузка компонент-переходников;
- нет именования через URL.

К основным достоинствам CORBA можно отнести межъязыковую и межплатформенную поддержку. Хотя CORBA-сервисы и отнесены к достоинствам технологии CORBA, их в равной степени можно одновременно отнести и к недостаткам CORBA ввиду практически полного отсутствия их реализации.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте толкование понятия «архитектура» применительно к информационным системам.
2. В чем суть доменного подхода?
3. Назовите основные классификационные признаки ИС.
4. Укажите отличительные характеристики информационно-управляющих систем.
5. Перечислите основные элементы управляющих систем.
6. Каково назначение систем мониторинга и управления ресурсами?
7. Укажите отличительную особенность систем управления производством.
8. На какой эталонной модели базируется система управления доступом?
9. Укажите стили проектирования ИС.
10. Перечислите набор характеристик качества ПО.
11. Каковы особенности централизованной архитектуры?
12. Каковы особенности распределенной архитектуры?
13. Какие существуют виды распределенных архитектур ИС?
14. Укажите достоинства архитектуры «файл-сервер».
15. Каковы области применения многозвенной архитектуры?
16. Укажите основные технологии архитектуры Web-приложений.
17. Каковы достоинства и недостатки технологии EJB?
18. Каковы достоинства и недостатки технологии DCOM?
19. Каковы достоинства и недостатки технологии CORBA?

Глава 2
АРХИТЕКТУРНЫЕ СТИЛИ

2.1. Понятие архитектурного стиля. Классификация архитектурных стилей

Архитектура может соответствовать некоторому архитектурному стилю.

Большинство архитектур построены на основе систем, использующих похожие решения. Сходство может быть определено как архитектурный стиль, который, в свою очередь, можно рассматривать как особый вид паттерна (шаблона). Архитектурный стиль представляет собой кодификацию опыта проектирования ИТ-систем. Примеры архитектурных стилей включают распределенный стиль, стиль «каналы и фильтры», стиль с централизованной обработкой данных, стиль, построенный на правилах, и т.д. Конкретная система может демонстрировать более одного архитектурного стиля.

Архитектурный стиль можно определить как семейство систем в терминах шаблона организации структуры. Точнее, архитектурный стиль определяет номенклатуру компонентов и типов соединительных звеньев, а также набор условий, в соответствии с которыми они могут соединяться. Архитектурный стиль определяется набором типов компонентов, во время счета выполняющих некоторую функцию, топологической раскладкой компонентов с указанием их взаимосвязей во время выполнения, набором семантических ограничений, набором соединителей, служащих средой сообщения, координации и сотрудничества между компонентами.

Иногда вместо термина архитектурный стиль используется термин ***архитектурный паттерн*** (шаблон). Однако следует отметить наличие принципиальных различий между архитектурными стилями и паттернами. Паттерн — это, по существу, фрагмент кода на конкретном языке программирования, а архитектурный стиль — это подход к проектированию.

Несмотря на многочисленные попытки до сих пор отсутствуют стандартные языки описания архитектур.

Принято выделять двенадцать базовых архитектурных стилей, которые делятся на пять групп (рис.2.1):

- потоки данных (Data Flow Systems);

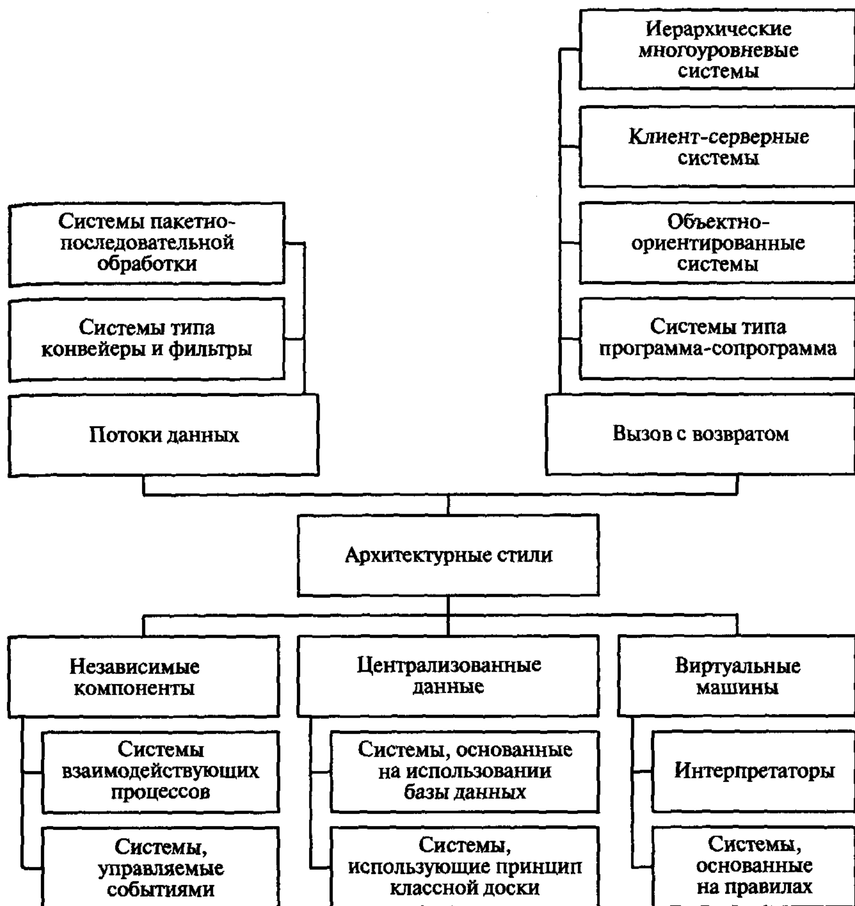


Рис. 2.1. Классификация архитектурных стилей

- вызов с возвратом (Call-and-Return Systems);
- независимые компоненты (Independent Component Systems);
- централизованные данные (Data-Centric Systems);
- виртуальные машины (Virtual Machines).

К системам, работающим по принципу потоков данных, относят системы двух архитектурных стилей: системы пакетно-последовательной обработки (Batch Sequential Systems) и системы типа конвейеры и фильтры (Pipe and Filter Architecture).

К *вызову с возвратом* относят четыре архитектурных стиля: программа-сопрограммы (Main Program and Subroutines), объектно-ориентированные системы (Object-Oriented Systems), клиент-серверные системы (Client-Server Systems), иерархические многоуровневые системы (Hierarchically Layered Systems).

К *системам, работающим по принципу независимых компонентов*, относят системы двух стилей взаимодействующих процессов (Communicating Sequential Processes) и системы, управляемые событиями (Event-Based Systems).

К *системам, использующие централизованные репозитария данных*, относят системы, основанные на использовании централизованной базы данных (Database Systems), и системы, использующие принцип классной доски (Blackboard Systems).

К *системам, работающим по принципу виртуальной машины*, относят интерпретаторы (Interpreters) и системы, основанные на правилах (Rule-Based Systems).

2.2. Потоки данных, вызов с возвратом

Системы, работающие по принципу потоков данных. К ним относят системы пакетно-последовательной обработки и системы типа конвейеры и фильтры.

Системы *пакетно-последовательной обработки* представляют собой набор связанных программных модулей, образующих линейную структуру. Задача делится на отдельные подзадачи. При этом выходные данные, сформированные одной подзадачей, используются в качестве входных данных для другой подзадачи. Обычно подзадачи выполняются последовательно. Данные могут передаваться либо через системную память, либо через внешние файлы. Для управления вычислительным процессом обычно используются скриптовые языки. Типичным примером такого подхода являются программы, написанные на языке Unix shell.

Архитектурный стиль *конвейеры и фильтры* близок к системам пакетно-последовательной обработки и может рассматриваться как обобщение пакетно-последовательной обработки. Система, построенная с использованием стиля конвейеры и фильтры, представляет собой множество модулей, каждому из которых ставится в соответствие один или несколько процессов. Модули могут быть как одинаковыми, так и разными и могут выполняться как на одном, так и на различных хостах. Данные с выходов одного модуля могут поступать на входы одного или нескольких других модулей. Система работает по принципу конвейера. Данные между отдельными ступенями конвейера могут передаваться разными способами, в частности, посредством использования механизмов межпроцессорного взаимодействиями, такими как pipe в Unix. Обработка носит преимущественно линейный характер, хотя конвейеры могут иметь обратные связи.

Примером данного подхода может служить компилятор. На вход компилятора поступает исходный код компилируемой программы. Функции первого фильтра выполняет лексический анализатор.

В качестве второй ступени выступает семантический анализатор, в качестве третьей ступени — оптимизатор, в качестве четвертой — генератор кода. Данный стиль широко используется при построении систем обработки сигналов и изображений.

Системы, работающие по принципу вызова с возвратом. К ним обычно относят системы, построенные по принципу программа-сопрограммы, клиент-серверные системы, объектно-ориентированные системы и иерархические многоуровневые системы. Отличительной особенностью систем, относящихся к данной группе, является то, что это синхронные программные архитектуры, в которых клиентская программа приостанавливает свое функционирование на то время, пока поставщик сервиса обслуживает запрос. По завершении обработки запроса сервер возвращает результат вычислений клиенту. Архитектуры, работающие по принципу вызова с возвратом, могут иметь произвольное число уровней вложенности.

Самым старым архитектурным стилем, относящимся к данной группе, является архитектурный стиль типа *программа-сопрограммы*. Программа выполняет функции контроллера, который управляет вычислительным процессом, в то время как функциональность реализуется в сопрограммах. Сопрограммы могут выполняться как на локальном, так и на удаленном хосте. В последнем случае речь идет о вызове удаленных процедур. Отличительной особенностью данного стиля является то, что программа имеет только одну нить управления. Данный стиль является, по существу, реализацией идей структурного программирования. В качестве разновидности стиля программа-сопрограммы можно выделить архитектуры типа *ведущий-ведомый* (Master-Slave Architecture). Обычно такие архитектуры не выделяют в отдельный стиль и рассматривают как параллельную версию стиля программа-сопрограммы. Отличительной особенностью данных архитектур является то, что основная программа и сопрограммы работают одновременно (параллельно). В данном случае на основную программу (ведущий) возлагаются функции диспетчеризации процесса вычислений. Сопрограмма (ведомый) получает задание, выполняет его, а по завершению задания запрашивает ведомого о новом задании. Данная архитектура может реализовываться как в рамках многопроцессорных систем, так и в сетевой среде с произвольной топологией.

Клиент-серверные системы можно рассматривать как специальный случай стиля программа-сопрограммы. Основное различие состоит в том, что клиент и сервер находятся на разных хостах, хотя в принципе клиент и сервер могут работать на одном хосте. Клиент — это процесс, который формирует запрос на обслуживание. Сервер — это процесс, который реализует сервис. В простейшем случае клиент посылает серверу команды и ожидает окончания выполнения запроса. Результатом выполнения запроса могут быть либо данные, либо подтверждение выполнения команды. Большинство серверов рабо-

тает с множеством клиентов. Широко используемая разновидность клиент-серверных систем — транзакционные системы, к которым могут быть отнесены, например системы продажи билетов. В системах, ориентированных на выполнение транзакций, число и типы транзакций фиксированы. Серверы в клиент-серверных системах при увеличении числа запросов могут масштабироваться.

Принято выделять два типа клиент-серверных систем: с толстым и тонким клиентом. Толстым называют клиентское приложение, которое содержит наряду с кодом, отвечающим за представление данных, достаточно большой объем кода, реализующего бизнес-логику. Тонкий клиент содержит код, который реализует исключительно функции, связанные с представлением информации. Обычно тонкий клиент — это Web-браузер.

Каждый из этих подходов имеет собственные достоинства и недостатки. Например, при использовании тонкого клиента достаточно просто модифицировать код приложения, поскольку нет необходимости обновлять код на многочисленных клиентских хостах. Код сервера обычно организуется по принципу супервизор-рабочие процессы. Супервизор обслуживает единую точку доступа к сервисам. Рабочие процессы обрабатывают каждый конкретный запрос.

Обычно реализуется следующий алгоритм: сервер открывает «хорошо известный порт» — принимает запросы к открытому порту — при поступлении запроса передает его одному из рабочих процессов и ожидает следующего запроса.

Клиент-серверные архитектуры имеют несколько разновидностей. Большинство ранних версий клиент-серверных приложений имели двухслойную организацию, в которой на сервере размещался репозитарий данных (рис. 2.2). На сервер возлагаются функции, связанные с управлением данными, а на клиентской стороне реализуется логика приложения (бизнес-логика) и логика представления.

Трехслойная архитектура отличается от двухслойной тем, что бизнес-логика размещается на отдельном сервере. Процессы становятся более устойчивыми, поскольку работают независимо от клиентов и от серверов (рис. 2.3).

Иногда используется и четырехслойная архитектура, которая может включать в себя следующие слои: контроллер домена, Web-сервер, сервер приложений, сервер баз данных. Сервер может общаться с клиентом как с установлением соединения, например по протоколу TCP/IP, так и без установления соединения, например по протоколу UDP. Сервер может либо сохранять, либо не сохранять информацию об обслуживаемом клиенте. Информацию о клиенте в этом случае называют состоянием. Информация о состоянии может быть полезна, если сеанс работы с клиентом включает обмен несколькими сообщениями.

Объектно-ориентированные системы можно также в определенном смысле рассматривать как частный случай систем типа Основ-

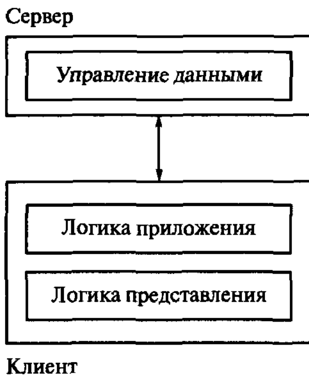


Рис. 2.2. Двухслойная организация клиент-серверной архитектуры

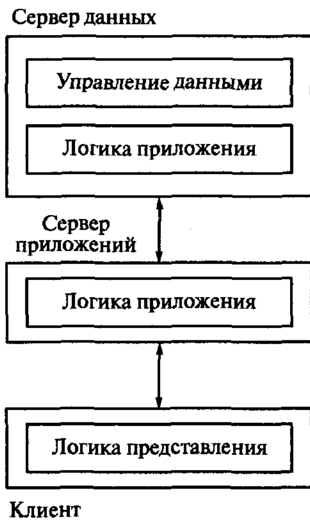


Рис. 2.3. Трехслойная организация клиент-серверной архитектуры

ная программа-сопрограммы, при этом объекты могут находиться как на одном, так и на разных хостах. Отличительной особенностью объектов является то, что объекты полностью инкапсулируют код и данные, и общение между объектами происходит либо посредством вызова процедур, либо посредством механизма сообщений.

Объектно-ориентированные системы также поддерживают механизмы наследования и/или делегирования. Пример простейшей объектно-ориентированной системы показан на рис. 2.4.

Одна из проблем, возникающих в процессе функционирования объектно-ориентированных систем, состоит в том, что объект должен знать, где находится тот объект, с которым он хочет взаимодействовать и какими интерфейсами он обладает. Привязка к объекту может осуществляться либо в статике, либо в динамике. Основное достоинство объектно-ориентированных систем — возможность сокрытия данных от пользователя. Это позволяет изменять внутреннее представление объекта, не уведомляя об этом клиента. Другим достоинством объектно-ориентированных систем является то, что они естественным образом поддерживают процесс распараллеливания вычислений.

Объектно-ориентированная система может рассматриваться как коллекция взаимодействующих агентов. Каждый объект при этом может быть как клиентом, так и сервером. Объекты могут располагаться в одном процессе, храниться в библиотеках, находиться в разных процессах, работающих на одном хосте и, наконец, располагаться на разных хостах. Важная разновидность объектов — компоненты,

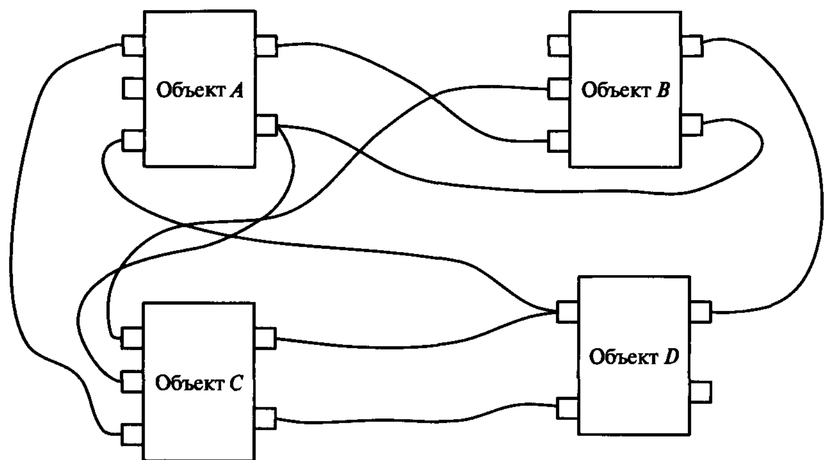


Рис. 2.4. Простейшая объектно-ориентированная система

которые являются объектами со специальными свойствами. Широко известны такие компонентные системы, как COM, CORBA, EJB. Типовая структура распределенной объектной системы показана на рис.2.5, где O1 — O5 — объекты.

Иерархические многоуровневые системы используются преимущественно для построения крупномасштабных приложений. Система содержит несколько слоев. Каждый слой можно рассматривать как виртуальную машину для вышележащего слоя. Каждый из слоев можно рассматривать также как набор сервисов для вышележащего слоя, таким образом, вышележащий слой работает в режиме клиента, а нижележащий — в режиме сервера. Обычно каждый из слоев взаимодействует с соседними слоями через четко определенные интерфейсы. Данный стиль повсеместно используется для построения стеков протоколов, в частности коммуникационных протоколов.

Рис. 2.5. Типовая структура распределенной объектной системы

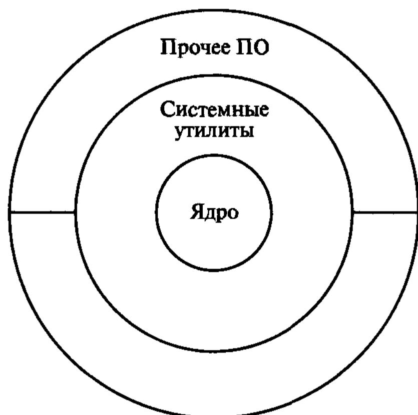


Рис. 2.6. Операционная система с трехуровневой организацией

Другим возможным применением данного стиля являются операционные системы. На рис. 2.6 показана структура операционной системы с трехуровневой организацией. Система включает три уровня: уровень ядра системы, уровень системных утилит и уровень, на котором работает прочее программное обеспечение. Данный подход был настолько популярен, что в процессорах фирмы Intel, начиная с модели Intel 286, заложена аппаратная поддержка работы четырехуровневой системой. Следует отметить, что современные ОС, такие как Unix и все ОС фирмы Microsoft, используют ОС, работающие только на двух уровнях, однако для построения стеков протоколов данный стиль продолжает широко использоваться. Основной недостаток состоит в том, что не все алгоритмы можно реализовать в виде многослойной архитектуры.

Основное достоинство данного стиля состоит в том, что он позволяет вести разработку кода для каждого из слоев независимо и иметь несколько вариантов реализации кода, кроме того, имеется возможность безболезненно модифицировать код послойно.

2.3. Независимые компоненты, централизованные данные

Системы, работающие по принципу независимых компонентов. К ним обычно относят системы взаимодействующих процессов и системы, управляемые событиями. Взаимодействующие процессы и системы, управляемые событиями, объединяет то, что в обоих случаях используется механизм неявного вызова оператора. Другими словами, вызывающий и вызываемый оператор могут существовать независимо и могут быть распределены по разным хостам. В систе-

мах, управляемых событиями, компоненты могут не знать о существовании друг друга. Активация оператора происходит по получению информации о наступлении события. Системы, управляемые событиями, могут основываться на использовании механизма «публикация-подписка».

В самом общем виде идея взаимодействующих процессов состоит в том, что имеется множество независимых процессов, которые обмениваются сообщениями. Эта модель была предложена и разработана еще в 1970-х гг. Хоаром и имеет множество вариантов реализаций, наиболее популярными из которых являются микроядерные архитектуры, параллельные системы, основанные на обмене сообщениями, а также мультиагентные системы.

Отличительной особенностью систем, управляемых событиями, является то, что процесс обработки запускается только тогда, когда происходит соответствующее событие. Можно привести следующие примеры событий: срабатывает таймер, пользователь перемещает мышь, пользователь щелкает клавишей мыши по пиктограмме или графическом объекте. Основное отличие систем, управляемых событиями, от систем взаимодействующих процессов состоит в том, что в первых получатель сообщения о событии не знает ничего об отправителе, а отправитель может ничего не знать о получателе или получателях информации о событии. Другими словами, системы, управляемые событиями, можно рассматривать как слабосвязанные системы. Системы, управляемые событиями, имеют много общего с программами, работающими от прерываний, поскольку прерывания и события по своей сути очень похожи тем, что им соответствует некоторое асинхронное событие, на которое система должна реагировать.

Обобщенная структура системы, управляемой событиями, показана на рис. 2.7. Основными компонентами системы являются источники событий, обработчики событий, коннекторы (диспетчер). Источники событий являются генераторами событий.

Источник генерирует событие в случае, если хочет сообщить другим компонентам о том, что произошло некоторое событие, которое может их заинтересовать. В типовом варианте получатель должен

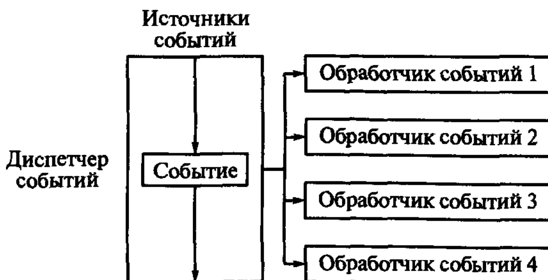


Рис. 2.7. Структура системы, управляемой событиями

зафиксировать свою заинтересованность в информации о конкретном событии. Диспетчер отвечает за три момента: сохранение сообщения о событии, если приемник не готов принять данное сообщение: маршрутизацию сообщений; запоминание информации о том, в какой информации заинтересован тот или иной компонент. Обычно источник и приемник не знают о существовании диспетчера. Обработчики событий — это программы, которые запускаются по приходу событий.

Отличительная особенность систем, управляемых событиями, состоит в том, что для них обычно невозможно указать поток управления.

Можно выделить, по крайней мере, три подстиля в рамках систем, управляемых событиями: централизованные системы, системы, используемые в промежуточном ПО, системы очередей сообщений.

Примером централизованных систем могут служить активные базы данных (поддерживающие работу с триггерами), системы разработки графических пользовательских интерфейсов.

В системах промежуточного ПО системы обработки событий используются достаточно активно. В частности, в CORBA имеется CORBA Event Service. Однако во многих реализациях она построена по централизованному принципу. В качестве одного из важнейших аспектов применения рассматриваемого архитектурного стиля являются очереди сообщений.

Системы, работающие по принципу централизованных данных (репозитария). К ним относят системы, основанные на использовании централизованной базы данных и системы, использующие принцип классной доски. Репозитарии иногда называют также системами с централизованным хранением данных (Data-centric systems).

Характерной особенностью архитектурных стилей, принадлежащих рассматриваемой группе, является наличие централизованного хранилища информации, содержимое которого общедоступно. Централизованное хранилище используется как для хранения данных, так и метаданных (знаний о данных). Централизованное хранилище обеспечивает удобный доступ к данным и метаданным.

Обобщенная структура системы, работающей по принципу репозитария, показана на рис. 2.8.

Рис. 2.8. Структура системы, работающей по принципу репозитария

Можно выделить следующие основные достоинства данного подхода: данные вводятся в систему однократно, что уменьшает затраты, исключает дублирование и уменьшает вероятность ошибки при вводе данных, пользователь может получить доступ к данным из множества разных приложений, репозитарий можно использовать как инструмент для обмена данными между приложениями, по мере необходимости репозитарий можно масштабировать.

Одним из наиболее часто используемых стилей является стиль, который известен как *централизованные базы данных*. Для организации разного рода систем хранения информации, которые можно использовать в качестве репозитариев, применяются разные подходы: информацию можно хранить в файлах в файловой системе; можно организовать централизованное хранилище, например, для хранения сериализованных объектов, можно хранить информацию в реляционных базах данных. Практически повсеместно используется последний подход. Можно выделить два основных типа баз данных: реляционные и объектно-ориентированные. Большинство используемых на практике баз данных относятся к реляционным. Информация, касающаяся баз данных, широко представлена в литературе и подробно рассматриваться не будет. Что касается баз знаний, в последние годы намечается устойчивая тенденция к использованию реляционных баз данных для хранения знаний.

Системы, использующие принцип *классной доски*, реализуют метафору классной доски, на которой можно писать мелом, при этом написанное доступно всем участникам обсуждения. В основе классная доска представляет собой базу данных, в которую можно записывать данные, читать их в ней и удалять из нее. Системы, построенные с использованием данного архитектурного стиля, находят применение, в первую очередь, в системах искусственного интеллекта, в качестве репозитария для хранения системных знаний.

Обобщенная структура системы, работающей по принципу классной доски, показана на рис. 2.9. В данной системе источники знаний представлены независимыми процессами, обладающими определен-

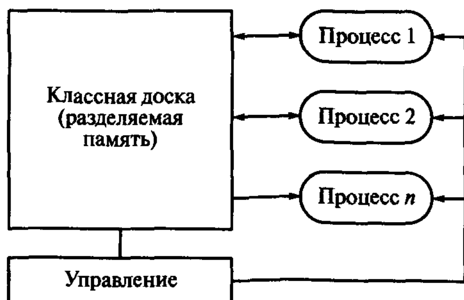


Рис. 2.9. Структура системы, работающей по принципу классной доски

ными знаниями о внешнем мире и о ходе решения текущей задачи. Процессы могут изменять данные, находящиеся на доске.

Классная доска представляет собой разделяемую память, в которой хранятся данные и знания о ходе решения задачи. Каждый процесс, выполнив свою часть работы по решению задачи, помещает результат на доску, где он становится доступен другим процессам. Каждый процесс может пользоваться услугами подсистемы информирования об изменении конкретной информации, помещенной на доску. Таким образом, если один процесс изменил данные, находящиеся на доске, то все процессы, которые заинтересованы в использовании этих данных, немедленно информируются о факте изменения данных. Уведомления процессам об изменении данных поступают процессам в форме сообщений.

В системах, работающих по принципу классной доски, для решения прикладных задач широко используются системы логического вывода, при этом используются три подхода: прямой логический вывод, обратный логический вывод, смешанные стратегии. Данный архитектурный стиль появился еще в 1970-х гг. и первоначально использовался в системах распознавания речи. В настоящее время данный стиль широко применяется, в частности, в мультиагентных системах для обеспечения взаимодействия агентов.

2.4. Виртуальные машины

К системам, работающим по принципу *виртуальной машины*, относят интерпретаторы и системы, основанные на правилах.

Виртуальная машина представляет собой эмулятор, который работает поверх аппаратной и (или) программной настройки и обеспечивает программный интерфейс, в общем случае отличный от программного интерфейса той платформы, на которой он работает.

Если рассматривать ИТ-систему как многослойную структуру, то виртуальная машина образует внешний слой, отвечающий за взаимодействие с клиентскими приложениями. Можно выделить следующие типовые варианты использования интерпретаторов: отладка программного обеспечения, предназначенного для работы на других, например контроллерных платформах; запуск программ, написанных для одной операционной системы (ОС) под управлением другой ОС, например для запуска приложений для Linux на платформе Windows; интерпретаторы команд; интерпретаторы некоторого языка, в качестве которого может выступать либо скриптовый язык, либо язык высокого уровня, либо некоторый специальный (доменно-ориентированный) язык.

Использование интерпретаторов в составе систем разработки важно прежде всего для разработки ПО встроенных систем. В этом случае интерпретатор представляет собой программную модель целе-

вой системы. Этот подход повсеместно используется при разработке и отладке программного обеспечения, как правило, для встроенных систем.

Использование интерпретатора в качестве средства переноса (портирования) приложений на другие платформы — достаточно популярный подход. В этой связи можно упомянуть такой повсеместно используемый продукт, как VMWare, который обеспечивает перенос приложений, в частности между Windows и Linux.

Интерпретаторы команд используются давно и повсеместно, например, используемая в DOS command.com и утилита Norton Commander. В Unix используются программы типа shell. В MS Windows интерпретатор команд встроен в Window Manager и управляется посредством мыши, а не вводом команды в виде строки. Широко используются такие интерпретаторы языков, как Basic, Perl, Python, а интерпретатор javascript встроен во все браузеры. Однако наибольшее практическое применение имеет виртуальная java машина (The Java Virtual Machine).

В системах, основанных на правилах, знания и логика представлены в виде множества правил — это отличает их от систем, основанных на использовании традиционных языков программирования. Структура типовой системы, ориентированной на работу с правилами, показана на рис. 2.10, включает в себя процессор логического вывода (движок), используемый для обработки правил; базу знаний; рабочую память и интерфейс пользователя.

Обычно пользователь получает доступ к системе через тот или иной пользовательский интерфейс. Запросы к системе выполняются с помощью специального языка.

Процессор логического вывода выполняет анализ запроса пользователя и определяет логический вывод. Во время выполнения вывода процессор по мере надобности выбирает из базы знаний правила и данные. В базе знаний хранятся два типа данных: факты, относящиеся к домену задач, и набор правил, также относящихся к домену задач. Процессор логического вывода применяет правила к данным до тех пор, пока не будет получен результат. Рабочая память используется для хранения промежуточных результатов.

Более сложные системы, основанные на правилах, могут содержать кроме перечисленных выше подсистем подсистему объясне-

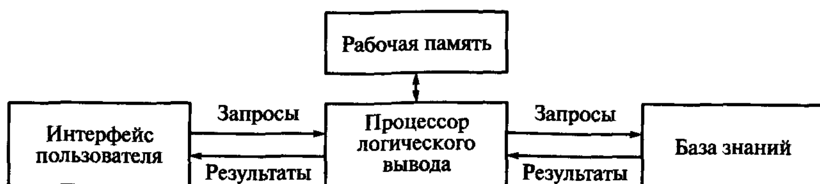


Рис. 2.10. Типовая система, ориентированная на работу с правилами

ния, которая представляет пользователю кроме конечного результата информацию о тех правилах, используемых в процессе логического вывода.

В системах, основанных на правилах, используется два типа логического вывода: прямой и обратный. Прямой логический вывод инициируется появлением нового факта, а в результате применения правил формируется некоторое заключение или выполняется некоторое действие. При использовании обратного вывода в качестве исходных данных выступает некоторое утверждение (цель), процесс логического вывода посредством применения правил подтверждает истинность исходного высказывания или его ложность. Возможно совместное применение прямого и обратного логического вывода. Что касается практического применения систем, основанных на правилах, то можно выделить три основных варианта их использования:

- оболочка экспертной системы, которая может наполняться фактами и правилами, например система CLIPS;
- библиотека компонентов, которая позволяет встраивать механизмы работы с правилами в пользовательские приложения, например система Jess;
- использование правил для управления бизнес-процессами, например система Jrules.

2.5. Использование стилей

Условия, при которых целесообразно использовать те или иные архитектурные стили, приведены в табл. 2.1.

Рассмотренные выше архитектуры можно отнести к «чистым» архитектурам. Они рассматривались в основном для объяснения при-

Таблица 2.1. **Условия использования архитектурных стилей**

Название стиля	Условия целесообразности использования
Системы пакетно-последовательной обработки	Задачу можно разделить на четко определенные подзадачи, каждая из которых использует единственную операцию ввода-вывода. Результаты операции либо отправляются пользователю, либо поступают на вход другой подзадачи. Имеются готовые приложения, которые можно использовать для решения отдельных подзадач
Системы типа конвейеры и фильтры	Алгоритм решения задачи можно представить как совокупность повторяющихся преобразований над однотипными и независимыми друг от друга наборами данных. Число ветвлений и обратных связей минимально

Название стиля	Условия целесообразности использования
Системы типа программа-сопрограммы	Порядок вычислений фиксирован, компоненты не могут делать ничего полезного, пока ждут результатов своих запросов к другим компонентам. Использование механизма наследования не дает существенных выгод
Объектно-ориентированные системы	Можно существенно уменьшить трудозатраты на разработку системы за счет использования в процессе разработки механизма наследования. Объекты находятся на разных хостах
Клиент-серверные системы	Задачу можно сформулировать как совокупность запросов с которыми клиенты обращаются к серверу, т. е. разделить на части, выделив клиентскую часть, отвечающую за формирование задания серверу, и серверную, отвечающую за выполнение запроса. Каждый клиент может обращаться к серверу в любое время. Сервер может только отвечать на запросы. Число клиентов может меняться, и они могут обращаться к серверу с разными запросами
Иерархические многоуровневые системы	Задачу можно представить в виде совокупности слоев с четко определенными интерфейсами, требуется иметь разные варианты бизнес-логики для разных слоев. Важна переносимость приложения между платформами, важна возможность использовать уже существующие реализации слоев
Системы взаимодействующих процессов	Передача сообщений — достаточный механизм взаимодействия для процессов. Нет большого объема долгоживущих централизованных данных
Системы, управляемые событиями	Система по своей природе асинхронная. Функционирование системы инициируется асинхронными событиями. Система может быть реализована как совокупность независимых процессов, функционирующих на разных платформах. Потребители событий отделены от сигнализаторов; важна масштабируемость в форме добавления процессов, переключаемых событиями

Название стиля	Условия целесообразности использования
Системы, основанные на использовании централизованной базы данных	Задачи можно разделить между выдающими запросы и отрабатывающими запросы или между производителями и потребителями данных Главный вопрос — хранение, представление, управление и поиск больших объемов связанных долгоживущих данных. Порядок исполнения компонентов определяется потоком входных запросов по доступу/обновлению данных; данные высокоструктурированы; доступна и экономична коммерческая СУБД
Системы, использующие принцип классной доски	Имеется большое число клиентов, которые общаются между собой для решения общей задачи. Важна масштабируемость в форме добавления потребителей данных
Интерпретаторы	Проектирование вычислений, когда нет реальной машины, или ее использование затруднено, требуется скрыть специфику платформы, пользователю необходимо предоставить возможность использовать либо стандартный, либо скриптовый язык программирования
Системы, основанные на правилах	Алгоритм решения задачи можно описать в терминах множества правил и способов их применения. Правила могут меняться конечным пользователем

роды каждого из стилей. Большинство реальных архитектур являются комбинацией нескольких стилей. Архитектурные стили в рамках конкретной архитектуры могут комбинироваться разными способами, основными из которых являются следующие: иерархии стилей; использование нескольких стилей на одном уровне иерархии; конкретное архитектурное решение, которое может быть описано в терминах двух или более архитектурных стилей.

Кроме того, конкретная архитектура может представлять собой комбинацию перечисленных выше подходов.

Первый подход предполагает, что компонент системы, построенной с использованием одного архитектурного стиля, внутри может быть организован с использованием другого стиля. Например, при использовании механизма Unix pipes каждый из программных модулей, работающих в составе конвейера, может реализовывать любой другой архитектурный стиль.

Второй подход разрешает использовать в рамках одного компонента несколько архитектурных стилей. Например, некоторый компонент может работать с репозитарием, используя одну часть своего интерфейса, а другую часть своего интерфейса использовать для взаимодействия с другими компонентами с помощью стиля конвейеры и фильтры. Другим примером могут служить «активные» базы данных, которые, с одной стороны, реализуют архитектурный стиль типа репозитарий, а, с другой стороны, реализуют неявный вызов. В системах, построенных по принципу классной доски, также часто реализуется такая комбинация стилей.

В ряде случаев одну и ту же архитектуру можно описать в терминах нескольких архитектурных стилей, например, систему Линда [17] можно рассматривать как систему взаимодействующих процессов, и как систему, использующую централизованный репозитарий.

На практике для достаточно сложных программных систем чаще всего применяется некоторая комбинация перечисленных выше подходов. Следует заметить, что для решения одной и той же задачи могут применять несколько разных архитектурных стилей. То же самое можно сказать и о семействах архитектур.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Охарактеризуйте понятие «архитектурный стиль».
2. Перечислите и охарактеризуйте основные архитектурные стили.
3. Перечислите и охарактеризуйте группы архитектурных стилей.
4. Охарактеризуйте архитектурный стиль пакетно-последовательная обработка, приведите примеры его использования.
5. Охарактеризуйте архитектурный стиль конвейеры и фильтры и приведите примеры его использования.
6. Охарактеризуйте архитектурный стиль программа-сопрограммы и приведите примеры его использования.
7. Охарактеризуйте архитектурный стиль объектно-ориентированные системы и приведите примеры его использования.
8. Охарактеризуйте архитектурный стиль клиент-серверные системы и приведите примеры его использования.
9. Охарактеризуйте архитектурный стиль иерархические многоуровневые системы и приведите примеры его использования.
10. Охарактеризуйте архитектурный стиль система взаимодействующих процессов и приведите примеры его использования.
11. Охарактеризуйте архитектурный стиль системы, управляемой событиями и приведите примеры его использования.
12. Охарактеризуйте архитектурный стиль системы, основанной на использовании централизованной базы данных, и приведите пример его использования.
13. Охарактеризуйте архитектурный стиль системы, использующей принцип классной доски, и приведите примеры его использования.

14. Охарактеризуйте архитектурный стиль интерпретаторы и приведите примеры его использования.
15. Охарактеризуйте архитектурный стиль системы, основанной на правилах и приведите примеры его использования.
16. Сформулируйте условия, при которых целесообразно использование каждого из рассмотренных архитектурных стилей.
17. Опишите основные способы совместного использования нескольких архитектурных стилей в рамках одной ИС.

Глава 3
ПАТТЕРНЫ И ФРЕЙМВОРКИ В АРХИТЕКТУРЕ ИС

3.1. Паттерны

Паттерны (шаблоны) впервые появились в середине 1990-х гг. как составная часть объектно-ориентированного подхода и рассматривались как набор объектов, организованных определенным образом для решения конкретного класса задач. Следует заметить, что хотя применительно к ИТ-системам паттерны используются немногим более 20 лет, в других отраслях, в первую очередь, в строительстве они успешно используются уже много десятков лет [3, 51, 52].

Обычно паттерн проектирования определяют как набор абстрактных классов, ориентированных на решение задач, относящихся к определенному домену. Паттерны проектирования иногда представляют в виде модели частичных решений, однако чаще их связывают с объектно-ориентированным программированием, хотя при работе с паттернами могут использоваться и другие парадигмы программирования. Следует заметить, что между объектами и паттернами нет жестких взаимных привязок и формирование концепции паттернов в рамках объектно-ориентированного подхода — это в определенной степени дело случая [3].

Значимость паттернов состоит и в том, что их использование позволяет выделить часто встречающиеся проблемы, дать им имена, предложить типовые решения, которые можно внедрять в создаваемые ИС. При использовании паттернов следует иметь в виду, что они не предоставляют готовый код. Функциональный код все равно приходится писать программисту.

Основное отличие паттернов от компонентов состоит в том, что компонент является модулем, который после настройки можно включить в состав системы, т.е. его можно рассматривать как готовый к употреблению строительный блок, а паттерн — это только заготовка, которую еще надо обработать, т.е. добавить код, определяющий функциональность.

Известны различные подходы к классификации паттернов. Так, паттерны предлагается классифицировать с точки зрения уровня абстракции на концептуальные, проектирования и программные.

Концептуальные паттерны — это паттерны, функционирование которых описывается в терминах предметной области. Такие паттерны относятся к приложению в целом или крупным подсистемам ИС.

Паттерны проектирования — это паттерны, для описания которых используются термины, относящиеся к разработке программных систем, такие как объект, класс, модуль. Паттерны проектирования описывают решение общих проблем в конкретном контексте.

Программные паттерны — это паттерны, для описания которых используются такие относительно низкоуровневые понятия как деревья, списки и т. п.

Используется также классификация, в соответствии с которой выделяются следующие типы паттернов: архитектурные, системные, структурные, поведенческие, производящие, параллельные программирования [15]. Кроме того, паттерны можно разделить на паттерны общего назначения и доменно-ориентированные паттерны. Паттерны общего назначения не привязаны явным образом к той или иной предметной области (домену), а доменно-ориентированные паттерны имеют такую привязку. В дальнейшем будут рассматриваться только паттерны общего назначения.

Архитектурный паттерн (architectural patterns) описывает структуру программной системы и определяет состав подсистем, их основные функции и допустимые способы компоновки подсистем. Архитектурные паттерны называют также архитектурными стилями. Архитектурные паттерны являются описанием, которое не зависит ни от платформы, ни от языка программирования. Архитектурные паттерны можно рассматривать как паттерны высокого уровня. Архитектурные стили были рассмотрены ранее.

Системные паттерны (system patterns) представляют собой приложение на верхнем (системном) уровне. Системные паттерны могут применяться в приложении для реализации типовых процессов и даже для поддержки взаимодействия разных приложений. Системные паттерны можно рассматривать как паттерны, использование которых позволяет получить улучшенные архитектурные решения.

Основные типы системных паттернов и их назначение приведены в табл. 3.1.

Структурные паттерны с одинаковой эффективностью применяются как для разделения, так и для объединения элементов приложения.

Основные типы структурных паттернов и их назначение приведены в табл. 3.2.

Структурные шаблоны могут быть использованы для решения различных задач. Например, шаблон Адаптер может обеспечить возможность двум несовместимым системам обмениваться информацией, тогда как шаблон Фасад позволяет отобразить упрощенный пользо-

Таблица 3.1. **Основные типы системных паттернов**

Типы паттернов	Назначение
Модель-Вид-Контроллер (Model-View-Controller (MVC))	Разделение компонента или подсистемы на три логические части (модель, представление и контроллер) для облегчения модификации или настройки каждой части в отдельности
Сессия (Session), Рабочая нить (Worker Thread)	Обеспечение возможности серверам в распределенных системах различать клиентов, что позволяет приложениям ассоциировать определенные состояния с клиент-серверными коммуникациями
Обратный вызов (Callback)	Обеспечение клиенту возможности регистрации на сервере для выполнения расширенных операций. Это позволяет серверу извещать клиента о завершении операции
Текущие обновления (Successive Update)	Обеспечение клиенту возможности постоянного получения обновлений от сервера. Такие обновления обычно отражают изменения данных сервера, впервые появившиеся или обновленные ресурсы либо изменения в состоянии бизнес-модели
Маршрутизатор (Router)	Отделение источников информации от ее получателей
Транзакция (Transaction)	Группирование коллекций методов таким образом, чтобы они либо были все успешно выполнены, либо все завершились неудачно

вательский интерфейс, не удаляя ненужных конкретному пользователю элементов управления.

Поведенческие паттерны (behavioral patterns) применяются для передачи управления в системе. Существуют методы организации управления, применение которых позволяет добиться значительного повышения как эффективности системы, так и удобства ее эксплуатации. Поведенческие шаблоны представляют собой набор проверенных на практике методов и обеспечивают понятные и простые в применении эвристические способы организации управления.

Основные типы поведенческих паттернов и их назначение приведены в табл. 3.3.

Производящие паттерны (creational patterns) предназначены для создания объектов в системе.

В ходе работы большинства объектно-ориентированных систем, независимо от уровня их сложности, создается множество экземпля-

Таблица 3.2. **Основные типы структурных паттернов и их назначение**

Типы паттернов	*Назначение*
Адаптер (Adapter)	Обеспечение взаимодействия двух классов путем преобразования интерфейса одного из них таким образом, чтобы им мог пользоваться другой класс
«Мост» (Bridge)	Разделение сложного компонента на две независимые, но взаимосвязанные иерархические структуры: функциональную абстракцию и внутреннюю реализацию. Это облегчает изменение любого аспекта компонента
Композит (Composite)	Предоставление гибкого механизма для создания иерархических древовидных структур произвольной сложности, элементы которых могут свободно взаимодействовать с единым интерфейсом
«Декоратор» (Decorator)	Предоставление механизма для добавления или удаления функциональности компонентов без изменения их внешнего представления или функций
«Фасад» (Façade)	Создание упрощенного интерфейса для группы подсистем или сложной подсистемы
«Приспособленец» (Flyweight)	Уменьшение количества объектов системы с многочисленными низкоуровневыми особенностями путем совместного использования подобных объектов
Полуобъект и протокол (Half-Object Plus Protocol)	Предоставление единой сущности, которая размещается в двух или более областях адресного пространства
Прокси (Proxy)	Представление другого объекта, обусловленное необходимостью обеспечения доступа или повышения скорости либо соображениями безопасности

ров объектов. Производящие паттерны облегчают процесс создания объектов, предоставляя разработчику следующие возможности:

• единый способ получения экземпляров объектов: в системе обеспечивается механизм создания объектов без необходимости идентификации определенных типов классов в программном коде;

• простота создания объектов: разработчик полностью избавлен от необходимости написания большого и сложного программного кода для получения экземпляра объекта;

• учет ограничений при создании объектов.

Таблица 3.3. **Основные типы поведенческих паттернов и их назначение**

Типы паттернов	Назначение
Цепочка ответственности (Chain of Responsibility)	Для организации в системе уровней ответственности, позволяет установить, должно ли сообщение обрабатываться на том уровне, где оно было получено, или же оно должно передаваться для обработки другому объекту
Команда (Command)	Обеспечивает обработку команды в виде объекта, что позволяет сохранять ее, передавать в качестве параметра методам, а также возвращать ее в виде результата, как и любой другой объект
Интерпретатор (Interpreter)	Определяет интерпретатор некоторого языка
Итератор (Iterator)	Предоставляет единый метод последовательного доступа к элементам коллекции, не зависящий от самой коллекции и никак с ней не связанный
Медиатор (Mediator)	Упрощает взаимодействие объектов системы путем создания специального объекта, который управляет распределением сообщений между остальными объектами
Моментальный «снимок» (Memento)	Сохраняет «моментальный список» состояния объекта, позволяющий такому объекту вернуться к исходному состоянию, не раскрывая своего содержимого внешнему миру
Состояние (State)	Предоставляет компоненту возможность гибкой рассылки сообщений интересующим его получателям
«Посетитель» (Visitor)	Обеспечивает простой и удобный в эксплуатации способ выполнения тех или иных операций для определенного семейства классов. Это достигается путем централизации с помощью данного шаблона возможных вариантов поведения, что позволяет модифицировать или расширять их, не затрагивая классы, на которые распространяются эти варианты поведения
Метод шаблона (Template Method)	Предоставляет метод, который позволяет подклассам перекрывать части метода, не прибегая к их переписыванию

Основные типы производящих паттернов и их назначение приведены в табл. 3.4.

Два из перечисленных выше паттерна, а именно Abstract Factory и Factory Method, базируются исключительно на концепции создания настраиваемых объектов. Подразумевается, что разработчик, применяющий их, при модернизации системы обеспечит механизм расширения создаваемых классов или интерфейсов. В силу этой особенности данные паттерны часто объединяются с другими производящими паттернами.

Паттерны параллельного программирования ориентированы на обеспечение корректного взаимодействия асинхронно протекающих процессов и ориентированы на решение двух основных задач (совместное использование ресурсов и управление доступом к ресурсам):

• совместное использование ресурсов. Если конкурирующие операции обращаются к одним и тем же данным или к общему ресурсу, то они могут конфликтовать друг с другом, если эти операции осуществляют доступ к ресурсу в один и тот же момент. Чтобы обеспе-

Таблица 3.4. **Основные типы производящих паттернов и их назначение**

Типы паттернов	Назначение
«Абстрактная фабрика» (Abstract Factory)	Обеспечивает создание семейств взаимосвязанных или зависящих друг от друга объектов без указания их конкретных классов
«Строитель» (Builder)	Упрощает создание сложных объектов путем определения класса, предназначенного для построения экземпляров другого класса. Шаблон Builder генерирует только одну сущность. Хотя эта сущность в свою очередь может содержать более одного класса, но один из полученных классов всегда является главным
«Метод фабрики» (Factory Method)	Определяет стандартный метод создания объекта, не связанный с вызовом конструктора, оставляя решение о том, какой именно объект создавать, за подклассами
«Прототип» (Prototype)	Облегчает динамическое создание путем определения классов, объекты которых могут создавать собственные дубликаты
«Одиночка» (Singleton)	Обеспечивает наличие в системе только одного экземпляра заданного класса, позволяя другим классам получать доступ к этому экземпляру

чить корректное выполнение таких операций, их нужно ограничить таким образом, чтобы доступ к общему ресурсу в конкретный момент времени получала только одна операция, однако чрезмерное ограничение операций может привести к их взаимной блокировке;

• управление доступом к ресурсам. Если операции получают доступ к общему ресурсу одновременно, возникает необходимость в том, чтобы они обращались к общему ресурсу в определенном порядке, например, объект не может быть удален из структуры данных до тех пор, пока данный объект не будет добавлен в некоторую другую структуру данных.

Основные типы паттернов параллельного программирования и их назначение показаны в табл. 3.5.

Для разных типов паттернов могут использоваться разные способы описания. Чаще всего используется описание, предложенное

Таблица 3.5. **Основные типы паттернов параллельного программирования и их назначение**

Типы паттернов	Назначение
Однопоточное выполнение (Single Threaded Execution)	Данный паттерн — это ключевой паттерн данной группы. Большинство задач, связанных с управлением доступа к разделяемым ресурсам, можно решить посредством использования этого паттерна
Охраняемая приостановка (Guarded Suspention)	Используется в том случае, когда поток имеет монопольный доступ к разделяемому ресурсу и оказывается, что он не может завершить выполнение операции над этим ресурсом, поскольку не имеет доступа к другим ресурсам
Объект блокировки (Lock Object)	Используется в том случае, когда требуется координировать доступ к нескольким ресурсам
Отмена (Balking)	Используется в том случае, когда операция должна быть выполнена либо немедленно, либо никогда
Планировщик (Scheduler)	Используется в ситуациях, когда важен порядок выполнения операций. Этот паттерн известен также под именем Диспетчер (Scheduler)
Блокировка чтения/записи (Read/Write Lock)	Обеспечивает альтернативный доступ к ресурсам, в случае, если одни операции могут совместно использовать тот или иной ресурс одновременно, а другие этого делать не могут
Производитель-потребитель (Producer/consumer)	Позволяет координировать объекты, создающие некоторый ресурс, и объекты, использующие этот ресурс

Типы паттернов	Назначение
Двухфазное завершение (Two-Phase Termination)	Применяется для правильного последовательного закрытия потоков
Двойная буферизация (Double Buffering)	Представляет собой специальную версию паттерна Производитель-потребитель. Этот паттерн позволяет создавать необходимый ресурс заранее
Асинхронная обработка (Asynchronous Processing)	Позволяет избежать ожидания результатов операции, если этот результат не нужен немедленно
Будущее (Future)	Позволяет классам, вызывающим операцию, не знать о том, является ли данная операция синхронной или асинхронной

в [3], в соответствии с которым полное описание паттерна выглядит следующим образом:

- название и тип;
- назначение;
- другие названия (если имеются);
- мотивация — какие проблемы можно решить с помощью данного паттерна;
- условия, при которых целесообразно применять данный паттерн;
- структура паттерна (в объектно-ориентированной нотации);
- объекты и паттерны, используемые в данном паттерне;
- результаты работы паттерна;
- рекомендации по применению;
- пример кода;
- примеры использования;
- родственные паттерны.

Как указывалось выше, паттерны — это классы проверенных практикой проектных решений, использование которых приводит к положительным результатам.

3.2. Антипаттерны

Антипаттерны (antipatterns), также известные как ловушки (pitfalls) — это классы наиболее часто внедряемых плохих решений проблем. Они изучаются как категория, в случае, когда их хотят из-

бежать в будущем, и некоторые отдельные случаи их могут быть распознаны при изучении неработающих систем [25].

Частью хорошей практики программирования является избегание антипаттернов.

Данная концепция также прекрасно подходит к машиностроению, строительству и другим отраслям. Несмотря на то что термин нечасто используется вне программной инженерии, концепция является универсальной.

Следует отметить, что общепринятой классификации антипаттернов не существует. Применительно к ИС можно выделить следующие типовые группы антипаттернов: в управлении разработкой ПО, антипаттерны в разработке ПО, в объектно-ориентированном проектировании, в области программирования, методологические и организационные антипаттерны.

Наиболее популярные *антипаттерны в управлении разработкой ПО* и их свойства приведены в табл. 3.6.

Антипаттерны в разработке ПО и их свойства приведены в табл. 3.7.

Антипаттерны в объектно-ориентированном проектировании и их свойства приведены в табл. 3.8.

Таблица 3.6. **Антипаттерны в управлении разработкой ПО и их свойства**

Типы антипаттернов	Свойства
«Дым и зеркала» (Smoke and mirrors)	Демонстрация того, как будут выглядеть ненаписанные функции. Название происходит от двух излюбленных способов, которыми фокусники скрывают свои секреты
«Раздувание» ПО (Software bloat)	Разрешение последующим версиям системы требовать все больше и больше ресурсов
«Функции для галочки»	Превращение программы в конгломерат плохо реализованных и не связанных между собой функций (как правило, для того чтобы заявить в рекламе, что функция есть)

Таблица 3.7. **Антипаттерны в разработке ПО и их свойства**

Типы антипаттернов	Свойства
Неопределенная точка зрения (Ambiguous viewpoint)	Представление модели без спецификации ее точки рассмотрения

Типы антипаттернов	Свойства
«Большой комок грязи» (Big ball of mud)	Система с нераспознаваемой структурой
«Бензиновая фабрика» (Gas factory)	Необязательная сложность дизайна
«Затычка на ввод данных» (Input kludge)	Забывчивость в спецификации и выполнении поддержки возможного неверного ввода
«Раздувание интерфейса» (Interface bloat)	Изготовление интерфейса очень мощным и очень трудным для осуществления
«Магическая» кнопка (Magic pushbutton)	Выполнение результатов действий пользователя в виде неподходящего (недостаточно абстрактного) интерфейса
«Перестыковка» (Re-Coupling)	Процесс внедрения ненужной зависимости
«Дымоход» (Stovepipe system)	Редко поддерживаемая сборка плохо связанных компонентов
«Гонки» (Race condition)	Непредвидение возможности наступления событий в порядке, отличном от ожидаемого

Таблица 3.8. **Антипаттерны в объектно-ориентированном проектировании и их свойства**

Типы антипаттернов	Свойства
Базовый класс-утилита (BaseBean)	Наследование функциональности из класса-утилиты вместо делегирования к нему
Вызов предка (CallSuper)	Для реализации прикладной функциональности методу класса-потомка требуется в обязательном порядке вызывать те же методы класса-предка
«Божественный» объект (God object)	Концентрация слишком большого количества функций в одной части системы (классе)

Типы антипаттернов	Свойства
«Полтергейст» (Poltergeist)	Объекты, чье единственное предназначение — передавать информацию другим объектам
«Проблема йо-йо» (Yo-yo problem)	Чрезмерная размытость сильно связанного кода (например, выполняемого по порядку) по иерархии классов
Синглетонизм (Singletonitis)	Избыточное использование паттерна Одиночка

Антипаттерны в области программирования и их свойства приведены в табл. 3.9.

Методологические антипаттерны и их свойства приведены в табл. 3.10.

Организационные антипаттерны и их свойства приведены в табл. 3.11.

Таблица 3.9. **Антипаттерны в области программирования и их свойства**

Типы антипаттернов	Свойства
Ненужная сложность (Accidental complexity)	Внесение ненужной сложности в решение
«Действие на расстоянии» (Action at a distance)	Неожиданное взаимодействие между широко разделенными частями системы
«Накопить и запустить» (Accumulate and fire)	Установка параметров подпрограмм в наборе глобальных переменных
«Лодочный якорь» (Boat anchor)	Сохранение неиспользуемой части системы
Активное ожидание (Busy spin)	Потребление ресурсов центрального процессора во время ожидания события, обычно при помощи постоянно повторяемой проверки, вместо того чтобы использовать систему сообщений
Кэширование ошибки (Caching failure)	Забывать сбросить флаг ошибки после ее обработки

Типы антипаттернов	Свойства
Инерция кода (Code momentum)	Сверхограничение части системы путем постоянного подразумевания ее поведения в других частях системы. Кодирование путем исключения (Coding by exception): Добавление нового кода для поддержки каждого специального распознанного случая
«Таинственный» код (Cryptic code)	Использование аббревиатур вместо мнемоничных имен
Жесткое кодирование (Hard code)	Внедрение предположений об окружении системы в слишком большом количестве точек ее реализации
Мягкое кодирование (Soft code)	Патологическая боязнь жесткого кодирования, приводящая к тому, что настраивается все что угодно, при этом конфигурирование системы само по себе превращается в программирование
«Поток лавы» (Lava flow)	Сохранение нежелательного (излишнего или низкокачественного) кода по причине того, что его удаление слишком дорого или будет иметь непредсказуемые последствия
«Магические» числа (Magic numbers)	Включение в алгоритмы чисел без объяснений их смысла
Процедурный код (Procedural code)	Когда другая парадигма является более подходящей
«Спагетти-код» (Spaghetti code)	Код с чрезмерно запутанным порядком выполнения
«Мыльный пузырь» (Soap bubble)	Класс, содержащий никогда не используемые данные

Таблица 3.10. **Методологические антипаттерны и их свойства**

Типы антипаттернов	Свойства
Программирование методом копирования-вставки (Copy and paste programming)	Копирование (и легкая модификация) существующего кода вместо создания общих решений

Типы антипаттернов	Свойства
Дефакторинг (Defactoring)	Процесс уничтожения функциональности и замены ее документацией
«Золотой молоток» (Golden hammer)	Сильная уверенность в том, что любимое решение универсально применимо. Название происходит от английской поговорки «когда в руках молоток, все проблемы кажутся гвоздями»
Фактор невероятности (Improbability factor)	Предположение о невозможности того, что сработает известная ошибка
Преждевременная оптимизация (Premature optimization)	Оптимизация на основе недостаточной информации
«Изобретение колеса» (Reinventing the wheel)	Ошибка адаптации существующего решения
«Изобретение квадратного колеса» (Reinventing the square wheel)	Создание плохого решения, когда существует хорошее

Таблица 3.11. **Организационные антипаттерны и их свойства**

Типы антипаттернов	Свойства
«Аналитический паралич» (Analysis paralysis)	Выделение непропорционально больших усилий в фазе анализа проекта
«Дойная корова» (Cash cow)	Закрытый продукт, который хорошо продается. Отсутствует стимул к его совершенствованию
Продолжительное устаревание (Continuous obsolescence)	Непропорционально большие усилия, направленные на перенос системы в новые условия эксплуатации
«Ползущее» совершенствование характеристик (Creeping featurism)	Добавление новых улучшений в ущерб качеству системы

Типы антипаттернов	Свойства
«Разработка комитетом» (Design by committee)	Имеется много участников разработки, но не имеется единого видения проекта
«Эскалация обязательств» (Escalation of commitment)	Продолжение реализации решения в том случае, когда неправильность его доказана
«Я тебе это говорил» (I told you so)	Когда игнорируется предупреждение эксперта, являющееся оправданным
Управление, основанное на числах (Management by numbers)	Уделяется избыточное внимание численным критериям управления, когда они неважны или стоимость их получения слишком высока
«Драконовские меры» (Management by perkele)	Жестко авторитарный стиль управления, в том случае, когда он не оправдан
«Управление грибами» (Mushroom management)	Удержание работников в неинформированном и занятом состоянии
«Расползание рамок» (Scope creep)	Неоправданное расширение объектов проекта без должного контроля
«Замкнутость на продавце» (Vendor lock-in)	Изготовление системы, жестко привязанной к одному поставщику
«Единственный знающий человек» (Single head of knowledge)	Единственный человек во всей организации контролирует жизненно важную область или информацию о проектируемой системе. Система оказывается «завязанной» на этого человека. При его уходе или бездействии работа останавливается
«Рыцарь на белом коне» (Knight in shining armor)	Личность, которая не совершает ошибок и пытается «починить» все, не сообщая, какие изменения он сделал или намерен сделать

3.3. Фреймворки

Термин «фреймворк», или «каркас», является достаточно популярным в среде разработчиков ИС и может обозначать разные понятия. В настоящее время нет общепринятого определения термина. Чаще

всего ***фреймворк*** определяют как набор типовых решений, методик проектирования и классов, которые могут быть использованы при решении множества сходных задач. В самом широком смысле под фреймворком понимают общепринятые архитектурно-структурные решения и (или) подходы к проектированию ИС. Применительно к программным системам фреймворк, или каркас, представляет собой набор классов или структур, которые описывают решение некоторого класса задач. Для решения конкретной задачи разработчик выполняет настройки соответствующих компонентов, собирает отдельные компоненты в систему и генерирует исполняемый код.

Фреймворки и паттерны имеют много общего и, в первую очередь, — это подходы к повторному использованию кода. Различие между паттернами и фреймворками состоит в следующем. Фреймворк можно рассматривать как реализацию системы паттернов проектирования (поведенческих паттернов). В то время как фреймворк — это исполняемая программа, а паттерн — это знание и опыт как решать конкретную задачу.

Еще одно отличие фреймворка от паттерна состоит в том, что фреймворк представляет собой скелетное решение достаточно крупной задачи и обычно включает в себя большое количество как паттернов, так и компонентов. Кроме того, фреймворки могут использовать подклассы и другие механизмы.

Фреймворки можно классифицировать по месту применения в ИТ-системе, способу использования и масштабу применения [45]. Классификация фреймворков приведена на рис. 3.1.

По месту применения фреймворки можно разделить на три типа:

• инфраструктурные фреймворки (System Infrastructure Frameworks);

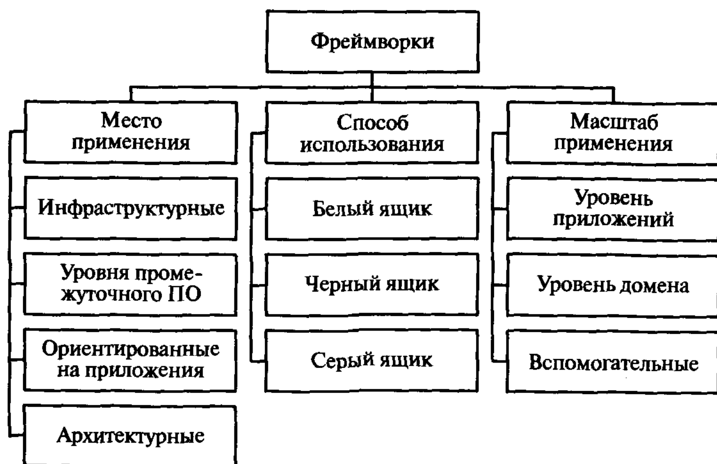


Рис. 3.1. Классификация фреймворков

• фреймворки уровня промежуточного ПО (Middleware Frameworks);

• фреймворки, ориентированные на приложения, относящиеся к конкретному предметному домену (Enterprise Application Frameworks).

• архитектурные фреймворки.

Использование *инфраструктурных фреймворков* упрощает разработку инфрастуктурных элементов, таких как, например операционные системы. Обычно такие фреймворки используются внутри организации и не поступают в продажу.

Фреймворки уровня промежуточного ПО используются для интеграции распределенных приложений и компонентов.

Фреймворки, ориентированные на приложения, используются, в первую очередь, для поддержания процесса разработки систем, ориентированных на конечного пользователя и принадлежащих некоторому конкретному предметному домену.

Международный стандарт ISO/IEC 42010 определяет *архитектурный фреймворк* как «совокупность соглашений, принципов и практик, используемых для описаний архитектур и принятых применительно к некоторому предметному домену и (или) в сообществе специалистов (заинтересованных лиц)»[14].

Типовой архитектурный фреймворк включает в себя следующие основные элементы: типовые для домена заинтересованные лица; типовые для домена проблемы; архитектурные точки зрения; правила интеграции различных точек зрения.

Можно выделить два типовых подхода к использованию фреймворков:

• по принципу белого ящика;

• по принципу черного ящика; кроме того, возможно использование гибридных подходов, которые называют серым ящиком.

Фреймворки, используемые по принципу белого ящика, называют также архитектурными фреймворками (architecture-driving framework). Эти фреймворки применяют наследование и динамическое связывание для формирования скелета приложения. Фреймворк, работающий по принципу белого ящика, определяется через интерфейсы объектов, которые разработчик может добавлять в систему. Основным недостатком данного подхода является то, что для того чтобы работать с фреймворками, необходимо иметь подробную информацию о классах, которые будут расширяться.

Фреймворки, используемые по принципу черного ящика, называют также фреймворками, управляемыми данными. При использовании фреймворков данного типа в качестве основных механизмов формирования приложения выступают композиция компонентов и параметризация. При этом требуемая функциональность достигается посредством добавление в фреймворк дополнительных компонентов. В общем случае работать с фреймворками, использующими

принцип черного ящика, проще, чем с фреймворками, реализующими принцип белого ящика, но их разработка является более сложной.

Большинство реальных фреймворков используют комбинацию рассмотренных выше подходов и их называют фреймворками, используемыми по принципу серого ящика (grey-box).

По масштабу применения фреймворки можно разделить на три группы: фреймворки уровня приложений, фреймворки уровня домена (организации), вспомогательные фреймворки.

Фреймворки уровня приложения (application frameworks) обеспечивают полный набор функций, которые реализуются типовыми приложениями. Обычно сюда входят GUI, базы данных, документация. Примером таких фреймворков могут быть MFC (Microsoft Foundation Classes), которые служат для создания приложений, ориентированных на работу в среде MS Windows.

Фреймворки уровня домена (Domain frameworks) используются для создания приложений, относящихся к определенному предметному домену.

В качестве домена может выступать как информационная система, включающая несколько взаимодействующих между собой приложений, например, системы сбора и обработки телеметрической информации, поступающих от сложной технической системы, так и целой организации, в качестве которой могут выступать, например, промышленные корпорации, органы государственной власти, правительственные ведомства и т. п.

В последнем случае речь идет о фреймворках уровня организации (enterprise). Термин «организация» понимается в самом широком смысле и включает коммерческие и некоммерческие организации, целые корпорации и их подразделения, различного рода ассоциации типа совместных предприятий и т.д. Следует особо отметить, что термин «организация» включает в себя такие элементы, как людей, собственно бизнес, информацию, технологии, а не только информационную систему.

Фреймворки уровня домена можно классифицировать по следующим признакам (рис. 3.2):

- назначению;
- принципам построения;
- гибкости при использовании;
- условиям распространения.

Возможны различные концептуальные подходы к построению фреймворков, в частности, ключевым элементом фреймворка может быть онтология. В основу фреймворка может быть положен тот или иной процесс разработки или ориентация на данные (данноцентрический подход). Фреймворк может быть ориентирован на эффективную поддержку сетевых взаимодействий.

Вспомогательные фреймворки (Support frameworks) ориентированы на решение частных задач, таких как управление памятью или файловой системой.

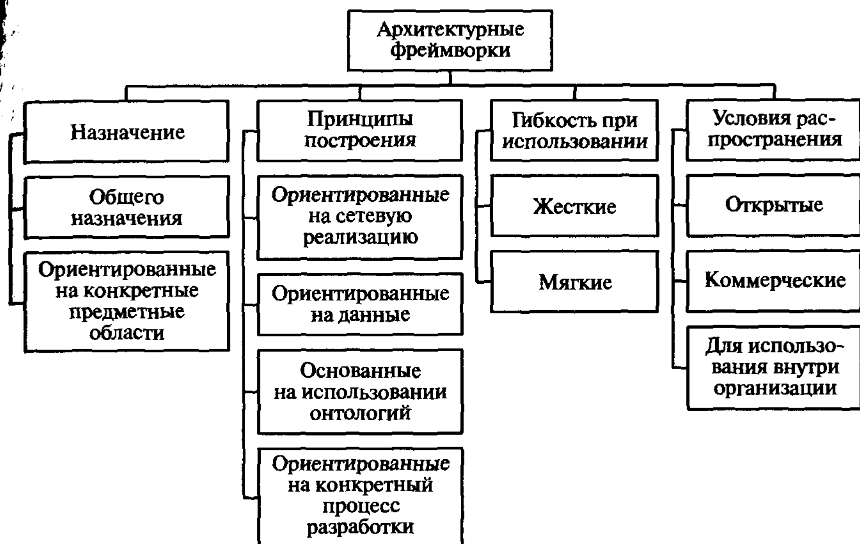


Рис. 3.2. Классификация фреймворков уровня домена

Фреймворки могут распространяться на коммерческой основе либо быть бесплатными для некоммерческого использования. Кроме того, фреймворк может быть ориентирован на использование внутри организации.

С точки зрения возможности реконфигурирования и возможности настройки на конкретное применение фреймворки можно разделить на «жесткие» и «гибкие». Жесткие фреймворки не предусматривают возможности настройки, а гибкие — разрешают настраивать фреймворк для решения конкретной задачи. Жесткие фреймворки могут требовать использовать конкретный инструментарий и конкретную методологию проектированию.

3.4. Примеры фреймворков

На данный момент существуют, по крайней мере, десятки различных фреймворков доменного уровня. Ниже в качестве примеров рассмотрены три фреймворка:

- фреймворк Захмана;
- фреймворк TOGAF;
- фреймворк министерства обороны США.

Это достаточно широко известные и активно используемые фреймворки, для построения которых использованы разные подходы.

Фреймворк Захмана. Это один из самых старых архитектурных фреймворков. Он назван по фамилии его создателя Джона Захмана

(John Zachman), который разработал данный подход еще в 1980-х гг., работая в IBM. С момента его создания появилось несколько вариантов данного фреймворка. Последний фреймворк Захмана (версия 2) Framework for Enterprise Architecture — был разработан компанией Zachman International в 2008 г. и анонсировался авторами как отраслевой стандарт [58].

В основе данного фреймворка лежит классификация (таксономия) артефактов, в состав которых входят такие категории, как данные и функциональность, а также модели, спецификации и документы. Фактически фреймворк Захмана в современном виде представляет собой онтологию верхнего уровня, которую разработчик конкретной ИС может расширять и уточнять, получая в результате онтологию, описывающую конкретную систему.

В основе предложенной классификации (таксономии) лежит идея, состоящая в том, что функционирование организации можно описать в терминах ответа на шесть простых вопросов: что, как, где, кто, когда, почему:

- используемые данные (что?);
- процессы и функции (как?);
- места выполнения процессов (где?);
- организации и персоналии (кто?);
- управляющие события (когда?);
- цели и ограничения, определяющие работу системы (почему?).

Ответы на эти вопросы можно давать с использованием различных понятий, т.е. с разной степенью детализации. При этом выделяется шесть уровней:

- уровень контекста;
- уровень бизнес-описаний;
- системный уровень;
- технологический уровень;
- технический уровень;
- уровень реальной системы.

Таким образом, формируется матрица размером 6×6, в каждой клетке которой ставятся в соответствие модели и артефакты (рис. 3.3).

Обозначенные шесть вопросов определяют шесть аспектов рассмотрения с точки зрения разных заинтересованных лиц (stakeholders), в число которых входят:

- аналитики;
- топ-менеджеры;
- архитекторы;
- разработчики;
- администраторы;
- пользователи.

В рамках рассматриваемого фреймворка определены следующие правила заполнения ячеек:

	Используемые данные (что?)	Процессы и функции (как?)	Места выполнения процессов (где?)	Организации и персоналии (кто?)	Управляющие события (когда?)	Цели и ограничения (почему?)	
Контекст	Список основных сущностей	Основные бизнес-процессы	Территориальное размещение организации	Важные внешние организации	Список событий	Бизнес-стратегия	**Аналитики**
Бизнес-модель	Отношения между сущностями	Подробное описание бизнес-процессов	Система логистики	Модель потоков работ	Базовый график работ	Дерево целей. Бизнес-план	**Топ-менеджеры**
Системная модель	Концептуальные модели данных	Архитектура приложений	Архитектура распределенной системы	Интерфейсы пользователя	Модель работы с событиями	Бизнес-правила	**Архитекторы**
Технологическая модель	Физическая модель данных	Программно-аппаратная архитектура	Технологическая архитектура	Архитектура представления	Алгоритмы обработки событий	Правила обработки событий	**Разработчики**
Детальное описание	Спецификации форматов данных	Исполняемый код	Архитектура сети	Роли и права пользователей	Обработка событий с помощью прерываний	Алгоритмы работы системы	**Администраторы**
Функционирующая организация	Данные	Реализуемая функциональность	Функционирующая сетевая инфраструктура	Организационная структура организации	История функционирования системы	Реализуемые стратегии	**Пользователи**

Рис. 3.3. Матричное представление фреймворка Захмана

• колонки можно менять местами, но нельзя добавлять новые колонки и удалять имеющиеся;

• каждой колонке соответствует собственная модель;

• каждая из моделей, соответствующих столбцам, должна быть уникальна;

• каждая строка (уровень) представляет собой описание системы с точки зрения пользователя или группы пользователей, т.е. представляет собой отдельный вид;

• каждая из ячеек уникальна;

• каждая клетка содержит описание аспекта реализации системы в виде определенной модели или текстового документа;

• заполнение клеток должно проводиться последовательно «сверху вниз».

1. Первая строка (контекст) соответствует уровню планирования бизнеса в целом с учетом внешних факторов. На этом уровне вводятся общие понятия, определяющие функционирование организации, определяются ключевые бизнес-процессы (например, закупка сырья, сбыт продукции), описывается место нахождение объектов, например бизнеса, определяются организации, с которыми осуществляется взаимодействие (партнеры, конкуренты, вышестоящие организации, собственные подразделения и т.д.), определяется список событий, влияющих на функционирование организации, формулируется бизнес-стратегия.

Фактически данная строка определяет контекст всех последующих строк и отражает самый общий взгляд на организацию и отражает точку зрения бизнес-аналитика.

2. Вторая строка (бизнес-модель) предназначена для описания функционирования организации в бизнес-терминах.

Этот уровень отражает видение организации менеджерами верхнего уровня. Здесь, в частности, используются описания реализуемых в организации бизнес-процессов.

3. Третий уровень (системная модель) соответствует видению организации системным архитектором. На этом уровне бизнес-процессы описываются уже в терминах ИТ-систем, включая определение типов данных, правила их преобразования и обработки для реализации функциональности, определенной на уровне 2.

4. На четвертом уровне (технологическая модель) выполняется привязка данных и операций над ними к определенным программным и аппаратным платформам и инструментальным средствам. Четвертый уровень отражает взгляд на ИС с точки зрения разработчиков программных и аппаратных средств.

5. Пятый уровень (детальное описание) включает описания конкретных моделей оборудования, топологию сети, средства разработки и собственно готовый программный код. Этот уровень отражает видения системы с точки зрения администраторов (системный администратор, администратор баз данных и т.п.).

6. Шестой уровень (функционирующая организация) описывает функционирующую систему и представляет собой видение ИС конечным пользователем. На данном уровне присутствуют такие элементы, как руководство пользователя, фактические базы данных и т. п.

Если перемещаться по ячейкам таблицы по строкам сверху вниз, то можно проследить, каким образом осуществляется детализация отдельных аспектов описания системы.

Так, первая *колонка используемые данные* определяет используемые в системе данные и отвечает на вопрос «что?». На верхнем уровне имеем простое перечисление основных сущностей. На втором уровне данные сущности используются для построения семантической модели, которая обычно описывается в виде ER-диаграммы. На третьем уровне полученная модель приводится к нормализованной форме, определяются все атрибуты и ключи. Четвертый уровень представляет собой физическую модель данных в системе либо в случае использования объектно-ориентированного подхода иерархию классов. Пятый уровень включает описание модели на языке управления данными для формирования таблиц, готовые библиотеки классов, табличные пространства СУБД. На последнем уровне описываются фактические наборы данных, размеры реально занимаемого дискового пространства, статистика обращений и т. д.

Колонка процессы и функции, отвечающая на вопрос «как?», содержит описания последовательной детализации процесса перехода от миссии организации к описанию отдельных операций.

В частности, первому уровню соответствует простое перечисление ключевых бизнес-процессов. Второй уровень содержит описания бизнес-процессов, которые затем детализируются и представляются в виде множества операций над данными и архитектуры приложений (третий уровень), методов классов (четвертый уровень), программного кода (пятый уровень) и исполняемых модулей (шестой уровень). Начиная с четвертого уровня, рассмотрение ведется уже не в рамках организации целом, а по отдельным приложениям.

Колонка места выполнения процессов, отвечающая на вопрос «где?», определяет пространственное распределение компонент системы и сетевую инфраструктуру. На уровне планирования бизнеса оказывается достаточным определить расположение всех основных объектов. На следующем уровне эти объекты объединяются в модель со связями, характеризующими взаимодействие между ними. На третьем уровне осуществляется привязка компонент информационной системы к узлам сети. Четвертый уровень служит для определения физической реализации в терминах аппаратных платформ, системного программного обеспечения и промежуточного ПО. На пятом уровне определяются используемые протоколы и спецификации каналов связи. Шестой уровень описывает функционирование реализованной сети.

Колонка организации и персоналии, отвечающая на вопрос «кто?», определяет участников процесса. На первом уровне планирования бизнеса определяется список организаций-партнеров, подразделений организации и выполняемые ими функции. На следующем уровне приводится полная организационная диаграмма, а также могут быть определены общие требования к информационной безопасности. Далее последовательно определяются участники бизнес-процессов и их роли, требования к пользовательским интерфейсам, правила доступа к отдельным объектам, физическая их реализация на уровне кода.

Колонка управляющие события отвечает на вопрос «когда?», определяет временные параметры бизнес-процессов и работы системы. Детализация осуществляется сверху вниз, начиная от списка значимых для функционирования системы событий (первый уровень) и базового графика работ (второй уровень). На третьем уровне определяются модели работы с событиями. На следующем уровне определяются алгоритмы обработки событий. Пятый уровень определяет программную реализацию процесса обработки событий, а на 6-м уровне — история функционирования системы, представленная, например, в форме записей в лог-файлах.

Последняя *колонка цели и ограничения* отвечает на вопрос «почему?», описывает мотивации и задает порядок перехода от задач бизнеса к требованиям, воторые предъявляются к отдельным элементам системы. Исходной точкой является бизнес-стратегия, которая последовательно транслируется в дерево целей и бизнес-план, затем в правила и ограничения для реализации бизнес-процессов, а на четвертом уровне — в соответствующие приложения, необходимые для включения в состав информационных систем и, в дальнейшем, в их физическую реализацию.

Рассматриваемый фреймворк в явном виде не позволяет описывать поведение системы в динамике, поскольку каждый элемент таблицы не может содержать как описание существующего состояния («как есть»), так и целевого, а также всех промежуточных состояний, при этом модель не содержит средств для четкого разделения этих различных «временны́х срезов». Однако этот недостаток достаточно просто можно устранить, если перейти от одномерной к двухмерной модели в виде куба, содержащего множество временны́х срезов.

Более серьезный недостаток данного фреймворка состоит в отсутствии встроенного механизма распространения изменений между элементами таблицы, что приводит к необходимости вручную отслеживать изменения всех взаимосвязей, проверки актуальности и внесения изменений в модели и другие артефакты во всех потенциально «затрагиваемых» ячейках.

Данный фреймворк не накладывает ограничений на использование тех или иных аппаратных и программных платформ и инструментальных средств разработки.

Несомненными достоинствами фреймворка Захмана являются простота для понимания как техническими, так и нетехническими специалистами, возможность поддержки обсуждений сложных вопросов с использованием относительно небольшого количества нетехнических понятий, нейтральность относительно инструментальных средств.

Фреймворк Захмана распространяется на коммерческой основе.

Фреймворк TOGAF. Open Group Architecture Framework (TOGAF) является одним из наиболее широко известных и часто используемых архитектурных фреймворков. Данный фреймворк ориентирован на решения задач, связанных с проектированием, планированием, реализацией и управлением корпоративной архитектурой и представляет собой набор инструментов, использование которых позволяет разрабатывать широкий спектр архитектур различного назначения. Он позволяет описывать ИС как совокупность строительных блоков (модулей), определять способы их соединения с помощью прилагаемого инструментария, включает список рекомендованных к применению стандартов и платформ, которые могут быть использованы при реализации ИС.

Разработка TOGAF ведется в рамках Архитектурного Форума (Architecture Forum), который функционирует в составе известной организации The Open Group, распространяющей данный фреймворк бесплатно для некоммерческого использования. Разработка проводится начиная с середины 1990-х гг. На момент издания книги последней является версия 9, которая появилась в начале 2009 г. С последними версиями можно ознакомиться на официальном сайте [57].

Следует отметить, что TOGAF дает собственное определение архитектуры, в соответствии с которым архитектура ИС — «формальное описание системы, или детальный план системы на уровне компонентов и методология их реализации» или «структура компонентов, их взаимосвязи, принципы их реализации и эволюции». В то время как в соответствии с общепринятым определением (стандарт ANSI/IEEE 1471-2000) архитектура определяется как «описание организации системы в терминах компонентов их взаимосвязей между собой и с окружающей средой и принципы управлением их разработкой и развитием».

В рамках TOGAF определяются четыре архитектурных домена:

- бизнес-архитектура (архитектура бизнес-процессов) определяет бизнес-стратегию, систему управления организацией и ключевые бизнес-процессы;
- архитектура уровня приложений определяет интерфейсы отдельных приложений и способы их использования в процессе реализации бизнес-процессов в терминах бизнес-сервисов;
- архитектура уровня данных, описывает логическую и физическую организацию корпоративных данных;
- технологическая (техническая) архитектура описывает аппаратную, программную и сетевую инфраструктуру.

Основными компонентами TOGAF являются:

• методика ADM (Architecture Development Method), определяющая процесс разработки архитектуры;

• руководства и методики проектирования, используемые в рамках ADM;

• фреймворк архитектурного описания (Architecture Content Framework), представляющий собой детальную модель результатов разработки архитектуры, в терминах артифактов и архитектурные блоки;

• архитектурный континиум организации (Enterprise Continuum), представляющий собой репозитарий архитектурных артефактов и артефактов реализации;

• эталонные модели TOGAF (TOGAF Reference Models), включающая в себя техническую эталонную модель (Technical Reference Model, TRM) и интегрированную модель информационной инфраструктуры (The Integrated Information Infrastructure Model, III-RM);

• фреймворк, описывающий организацию (Architecture Capability Framework), описывающий структуру организации, персонал, роли и ответственности.

Ключевым элементом рассматриваемого фреймворка является методика ADM, в соответствии с которой процесс разработки архитектуры разбивается на 9 фаз (рис. 3.4). ADM реализует итерационный процесс проектирования на двух уровнях. На верхнем уровне для каждой итерации повторяются действия, определенные на каждой из фаз, а на нижнем уровне реализуются итерации внутри отдельных фаз. Все решения принимаются исходя из текущих требований бизнеса и имеющихся решений.

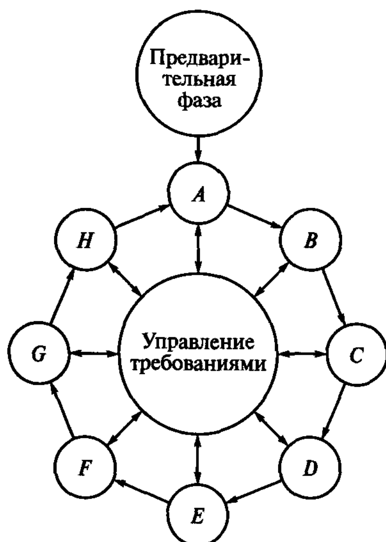


Рис. 3.4. Фазы разработки архитектуры

Характеристика фаз процесса разработки архитектуры приведена в табл. 3.12.

Каждая фаза, в свою очередь, разбивается на этапы (Steps),которые, в свою очередь, могут разбиваться на еще более мелкие шаги. Для каждого этапа определяются решаемые задачи, входные данные и выходные артефакты.

Например, фаза *B* включает следующие основные этапы:

• выбор эталонной модели, видов и инструментальных средств;

• формирование архитектурных описаний текущей и целевой архитектур;

Таблица 3.12. **Характеристика фаз процесса разработки архитектуры**

Фазы процесса	Характеристика
Предварительная фаза (Preliminary Phase)	Определение рамок разработки, уточнение модели в целях учета специфики организации и определение общих принципов реализации ИС; определение методов управления
Фаза *A*, разработка общего представления (Architecture Vision)	Разработка общего видения архитектуры, определение границ проекта, определение ограничений, утверждение плана работ и разработка заданий основным исполнителям
Фаза *B*, разработка бизнес-архитектуры (Business Architecture)	Определение общих принципов функционирования организации в терминых бизнес-процессов, принципов управления проектом и анализ того, в какой степени результаты деятельности организации соответствуют бизнес-целям. Основными артефактами данной фазы являются структуры организации, бизнес-цели, бизнес-функции, бизнес-сервисы, бизнес-процессы, основные роли, взаимосвязи между организационной структурой организации и реализуемыми функциями
Фаза *C*, разработка информационной архитектуры (Information Systems Architectures)	С точки зрения теории, сначала следует разрабатывать архитектуру данных, а затем архитектуру приложений, однако на практике не всегда удается реализовать такой подход, и обычно на данной фазе требуется выполнить несколько итераций
Фаза *D*, разработка технологической или технической архитектура (Technology Architecture)	Выбор аппаратных средств и сетевой инфраструктуры и механизмов их взаимодействия
Фаза *E*, возможности и решения (Opportunities and Solutions)	Начинается процесс реализации разработанной архитектуры: формируются задания соответствующим специалистам на реализацию, выбираются подходы к реализации, решается вопрос следует ли выполнять внедрение собственными силами, привлечь сторонние организации, закупить готовую систему, использовать коммерческое или свободно распространяемое ПО

Фазы процесса	Характеристика
Фаза *F*, планирование перехода к новой архитектуре (Migration Planning)	Осуществляется планирование перехода к новой системе, выполняются оценки рисков, прорабатываются детали перехода на новую архитектуру
Фаза *G*, формирование системы управления реализацией (Implementation Governance)	Формируется система управления преобразованиями: специфицируются проблемы, возникающие на этапе внедрения, осуществляется мониторинг процесса внедрения новой архитектуры
Фаза *H*, управление изменением архитектуры (Architecture Change Management)	Осуществляется управление изменением архитектуры: реализуется постоянный мониторинг процесса управления изменениями, отслеживается адекватность внесенных изменений разработанной архитектуре, отлаживается процесс управления архитектурными изменениями

• проведение анализа расхождений (gap analysis);

• разработку плана действий по переходу на новую архитектуру;

• проведение согласования со всеми заинтересованными сторонами;

• корректировку плана действий по результатам взаимодействия с заинтересованными сторонами;

• документирование архитектуры.

Основными сферами применения TOGAF являются крупные организации, при этом TOGAF может использоваться как единственный фреймворк, так и совместно с другими фреймворками, которые ориентированы на конкретные предметные области, такие как оборона, финансы, здравоохранение. В состав TOGAF, в частности, входит отдельный документ, поясняющий соответствие между понятиями TOGAF и фреймворком Захмана.

Если сравнивать TOGAF и фреймворк Захмана, то просматривается различие в подходах. Если фреймворк Захмана ориентирован на описание некоторой конкретной архитектуры и в его основе лежит таксономия, то в основе TOGAF лежит понятие архитектурного континиума, т. е. архитектура постоянно находится в процессе изменения, отслеживая изменения бизнес-требований, поэтому на первый план выходит ADM.

Архитектурный фреймворк министерства обороны США Department of Defense Architecture Framework (DoDAF). Фреймворк для министерства обороны США включает в себя набор точек зрения (viewpoints), отражающих взгляды на систему со стороны различных

заинтересованных сторон. Все вооружения и информационные системы, закупаемые МО США, должны документироваться в соответствии с данным фреймворком. Несмотря на то, что данный фреймворк ориентирован на системы военного назначения, он является полностью открытым и свободно распространяемым. Всю информацию по данному фреймворку можно найти на сайте МО США [56].

Первая версия DoDAF появилась в 2003 г., а на данный момент последней версией DoDAF была версия 2 от 2009 г.

На базе DoDAF был создан целый ряд других фреймворков, к числу которых относятся фреймворк НАТО (NATO Architecture Framework, NAF) и фреймворк МО Великобритании (Ministry of Defence (United Kingdom) Architecture Framework (MODAF)). DoDAF имеет много общих черт с TOGAF, в частности, одним из ключевых элементов как TOGAF, так и DoDAF является репозитарий, содержимое которого доступно всем заинтересованным сторонам.

Отличительной особенностью DoDAF является ориентация на данные, авторы позиционируют данный фреймворк как data centric, т. е., по мнению авторов, самым главным элементом ИС являются данные, а обрабатывающие их приложения являются производными. При создании семейств продуктов важно сохранить данные для повторного использования. Интересно отметить, что предыдущая версия DoDAF (версия 1.5) определяла данный фреймворк как ориентированный на сетевое взаимодействие (net centric).

Ориентация на данные определяет класс систем, на проектирование которых ориентирован DoDAF — это прежде всего системы сбора, хранения и обработки данных в целях их использования в процессе принятия решений.

Основными элементами архитектурного описания являются модели (models), виды (view) и точки зрения (viewpoints).

Модель определяется как шаблон (template) для сбора данных, при этом выделяются следующие типы моделей:

• таблицы, данные в которых представлены в виде строк и столбцов;

• графические изображения, описывающие структурные аспекты архитектурного решения;

• графические изображения, описывающие поведенческие аспекты архитектурного решения;

• отображения, описывающие взаимосвязь между двумя типами информации в форме матрицы;

• онтологии, являющиеся расширением онтологии, определенной в DoDAF;

• картинки в свободном формате;

• разного рода временные диаграммы.

Вид определяется как способ представления связанного набора данных в понятном пользователю виде. Это могут быть документы, таблицы, графики и т. п.

Точка зрения описывает данные, поступающие от одного или нескольких источников, которые организованы таким образом, чтобы были полезны при принятии решений и может включать несколько видов. Точка зрение представляет собой упорядоченное множество видов.

Архитектурное описание в рамках DoDAF определяется как множество видов, используемых для документирования архитектуры.

В рамках DoDAF V2.0 определяются восемь точек зрения:

• обобщенная точка зрения (All Viewpoint (AV), которая интегрирует все точки зрения и образует архитектурный контекст для других точек зрения;

• точка зрения, определяющая потенциальные возможности (Capability Viewpoint, CV), использующая такие понятия как, например, сроки поставки оборудования, обучения персонала и т.д.);

• точка зрения, определяющая данные и информацию (Data and Information Viewpoint, DIV), рассматривает структуры данных и способы их представления для разных целей;

• операционная точка зрения (Operational Viewpoint, OV) рассматривает систему с точки зрения сценариев работы, активностей;

• проектная точка зрения (Project Viewpoint PV), которая рассматривает систему с точки зрения требуемых характеристик и возможностей, а также с точки зрения процесса проектирования и других реализуемых проектов (эта точка зрения активно используется при реализации процесса закупок систем для нужд МО);

• сервисная точка зрения (Services Viewpoint, SvcV) рассматривает систему как совокупность взаимодействующих сервисов;

• точка зрения, учитывающая стандарты (Standards Viewpoint, StdV), в частности, действующие технические стандарты, методики, руководства, ограничения и т.п.;

• системная точка зрения (Systems Viewpoint, SV) рассматривает систему как совокупность взаимодействующих подсистем, рассматривает способы взаимодействия подсистем и используется преимущественно при необходимости работать с унаследованными системами.

DoDAF служит также руководством по разработке интегрированных архитектур, каждая из которых включает в себя множество видов. Однако при этом не требуется использовать именно предлагаемую методологию проектирования в процессе разработки архитектуры.

Термин «интегрированная архитектура» означает, что данные могут использоваться для формирования более чем одного архитектурного вида.

Данные могут собираться, организовываться и храниться с использованием различных программных продуктов и инструментов, в частности коммерческих.

Предлагаемая методика построения архитектурного описания включает в себя шесть шагов:

1) определение назначения архитектурного описания;

2) определение требуемого уровня детализации разрабатываемого описания;

3) определение того, какие данные необходимы для построения архитектурного описания;

4) сбор и обработка требуемых данных. Построение описания;

5) анализ архитектуры на соответствие требованиям;

6) документирование архитектуры и представление результатов в виде, удобном для практического применения в системах поддержки принятия решений.

Визуализация видов используется исключительно в иллюстративных целях.

Для успешного использования DoDAF пользователям предлагается руководствоваться восемью принципами, которые представляют собой концепции достаточно высокого уровня.

1) архитектурное описание должно быть четко ориентировано на провозглашенные цели;

2) архитектурное описание должно быть по возможности простым и понятным, но не упрощенным;

3) архитектурное описание должно облегчать, а не затруднять процесс принятия решений;

4) архитектурное описание должно быть составлено таким образом, чтобы его можно было использовать для сравнения различных архитектур;

5) при составлении архитектурного описания должны в максимальной степени использоваться стандартные типы данных, определенные в DM2;

6) архитектурное описание должно выполняться в терминах самих данных, а не инструментальных средств работы с данными;

7) архитектурные данные должны быть организованы в виде, удобном для групповой работы;

8) архитектурное описание должно быть построено таким образом, чтобы его можно было использовать в сетевой среде, т.е. должно быть сетецентричным.

В рамках DoDAF используется мета-модель данных, которая называется Data Meta-Model (DM2). Она представляет собой онтологию и имеет несколько уровней, каждый из которых отражает наиболее важный для конкретной группы пользователей уровень представления информации. Пользователь может расширять данную модель в соответствии со своими нуждами. В рамках DM2 определены три верхних уровня:

1) концептуальная модель данных (Conceptual Data Model (CDM));

2) логическая модель данных (Logical Data Model, LDM);

3) спецификация обмена данными на физическом уровне (Physical Exchange Specification, PES).

Данные модели являются составной частью рассматриваемого фреймворка.

Концептуальная модель дает возможность строить архитектурное описание в нетехнических терминах, которое понятно не только ИТ-специалистам.

Логическая модель является расширением концептуальной модели за счет добавления атрибутов к понятиям, определенным на концептуальном уровне.

Спецификация обмена данными представляет собой платформенно независимое средство, обеспечивающее обмен данными между разными моделями. Данная спецификция основана на использовании XML и XSD. Для каждой из моделей, входящей в состав DoDAF, имеется собственное XSD-описание.

Семантически близкие понятия сгруппированы в кластеры, основными из которых являются следующие:

- цели;
- средства достижения целей (что нужно предпринять в тех или иных обстоятельствах для достижения желаемого эффекта);
- активности (работы, которые выполняют некоторые преобразования или изменяют состояние);
- исполнители (то, что реализует активности, в частности, системы, персонал, организации);
- сервисы (бизнес-сервисы или программные сервисы);
- материальные потоки (описывают процессы взаимодействия между активностями);
- информация и данные;
- проекты (все формы планируемых активностей);
- трейнинги/квалификация/образование (описание требований с точки зрения квалификации персонала);
- правила (правила типа «как?», стандарты, соглашения, ограничения, относящиеся к архитектуре);
- метрики (все виды измерений, которые применимы к архитектуре);
- местоположение (все виды размещения, включая точки, линии, фигуры, место нахождения конкретного оборудования, URL и т.д.).

Если сравнивать DoDAF с рассмотренными ранее фреймворками Захмана и TOGAF, то следует отметить, что он существенно отличается по назначению, поскольку ориентирован не на разработку всей системы, а прежде всего на разработку архитектурного описания системы, в целях формирования задания на разработку с учетом ранее выполненных разработок и разработок, выполняющихся одновременно. В качестве конечного результата выступает не ИС, а только ее документированное архитектурное описание. Кроме того, DoDAF дает пользователю значительно большую свободу выбора инструментальных средств и моделей для решения конкретной задачи и активно предлагает ему выбирать только нужные модели.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте определение понятий «паттерн» и «фреймворк».
2. Приведите классификацию паттернов.
3. В чем состоит различие между между паттернами и фреймворками?
4. Перечислите и охарактеризуйте основные структурные паттерны.
5. Что такое антипаттерны?
6. Приведите классификацию антипаттернов.
7. Приведите классификацию фреймворков.
8. Охарактеризуйте фреймворк Захмана.
9. Какое место занимает онтология в фреймворке Захмана?
10. Перечислите достоинства и недостатки фреймворка Захмана и укажите условия, при которых целесообразно использование фреймворка Захмана.
11. Охарактеризуйте фреймворк TOGAF.
12. Какое место занимает методика ADM в фреймворке TOGAF?
13. Перечислите достоинства и недостатки фреймворка TOGAF и укажите условия, при которых целесообразно использования данного фреймворка.
14. Охарактеризуйте фреймворк DoDAF.
15. Почему фреймворк DoDAF определяют как data centric?
16. Перечислите достоинства и недостатки фреймворка DoDAF и укажите условия, при которых целесообразно использования данного фреймворка.
17. В чем состоит различие подходов, используемых в между фреймворках Захмана, TOGAF и DoDAF?

Глава 4

КОМПОНЕНТНЫЕ ТЕХНОЛОГИИ РЕАЛИЗАЦИИ ИНФОРМАЦИОННЫХ СИСТЕМ

4.1. Понятие компонента. Компонентные технологии

Термин «компонент» в ИТ-отрасли используется для обозначения различных понятий. Можно выделить аппаратные (hardware components) и программные (software components) компоненты.

Примерами *аппаратного компонента* могут служить микропроцессор, жесткий диск, модули памяти.

Термин *программный компонент* (ПК) используется для обозначения двух связанных, но разных понятий. Если речь идет о программной архитектуре, то обычно под компонентом подразумевается программный модуль, реализующий некоторую функцию или набор функций, который решает определенные подзадачи в рамках общих задач системы. Это те компоненты, которые могут быть изображены на диаграммах компонентов (component diagram) в языке UML. Если речь идет о компонентных технологиях программирования или компонентно-ориентированной (component based) разработке программного обеспечения (ПО), то под ПК понимают объекты со специальными свойствами. В дальнейшем термин компонент будет употребляться во втором значении.

В этом случае понятие ПК выступает в качестве ключевого для определения понятия таких понятий, как компонентно-ориентированное программирование и компонентно-ориентированный подход к проектированию ИС.

Согласно [55] ПК представляет собой структурную единицу программной системы с четко определенным интерфейсом, который полностью описывает ее зависимости от контекста.

ПК представляет собой откомпилированный автономный программный модуль. Его можно объединять с другими модулями или кодом, организованным другим способом для создания приложений. ПК может быть простым (кнопка) или сложным (компонент, реализующий управление сетью).

Чаще всего под ПК понимают откомпилированный «двоичный компонент», который можно интегрировать в приложение «на лету». При работе с компонентами исходный код обычно недоступен и по-

этому компонент нельзя изменять. Под данное определение подпадают, в частности, динамические библиотеки.

ПК используется для построения компонентно-ориентированных приложений. Каждое из приложений представляет собой среду, включающую средства «склеивания» или скелетный код, в который можно встраивать ПК. ПК, в свою очередь, могут взаимодействовать со средой и операционной системой, а также друг с другом. Изменение интерфейса ПК приводит к изменению его кода, но изменение способа реализации не обязательно приводит к изменению интерфейса.

ПК можно рассматривать и как пакет, ориентированный на повторное использование кода, который можно приобрести у независимого производителя, и как элемент системы, который обладает явно определенной функциональностью и может быть заменен на другой ПК. И, наконец, ПК можно определить как физическую реализацию некоторого набора интерфейсов.

Из сказанного выше понятно, что не существует общепринятого определения компонента.

ПК можно рассматривать с точки зрения реализуемой функциональности, реализации, исполняемого кода.

В первом случае речь идет об описании ПК в терминах реализуемых сервисов. Для этого обычно используются языки определения интерфейсов. Во втором случае описывается внутреннее устройство ПК. Для этого можно использовать, например диаграммы классов. В третьем случае описывается функционирование ПК с учетом специфики платформы.

Возникает естественный вопрос о том, как соотносятся понятия ПК и объект. На данный вопрос имеются разные точки зрения. Крайние точки зрения выглядят следующим образом. Одна точка зрения состоит в том, что ПК — это объект со специальными свойствами. Другая — в том, что ПК и объект — это разные сущности, а потому компонент не является объектом и представляет собой самостоятельную сущность. Наличие разных точек зрения определяется, с одной стороны, отсутствием общепринятого определение понятия ПК, а, с другой стороны, наличием разных компонентных технологий, которые существенно отличаются друг от друга с точки зрения используемых моделей компонентов.

Безусловно ПК и объекты имеют много общего, но есть и различия.

Концептуально объект представляет собой элемент модели реального мира, который имеет уникальный идентификатор и описывает определенный класс сущностей. Объект имеет состояние, которое определяется значениями внутренних переменных. Поведение объекта описывается через множество доступных методов. Состояние объекта может изменяться только посредством вызова его методов. На базе конкретного класса можно создавать произвольное число объектов. Объект может взаимодействовать с другими объектами

посредством вызова методов объекта-сервера, причем один и тот же объект может выступать как в качестве клиента, так и в качестве сервера.

Как ПК, так и объекты, поддерживают инкапсуляцию и доступ через «хорошо определенные интерфейсы». Обычно как объекты, так и ПК поддерживают «сильную» инкапсуляцию, при которой доступ к переменным (атрибутам) возможен только с использованием методов объектов. Как объекты, так и компоненты — это модели объектов реального мира. Часто принято считать, что ПК — это большие объекты, однако с этим трудно согласиться, поскольку объекты могут быть большими, а компоненты маленькими.

Типовой ПК обладает следующими свойствами:

- представляет собой фрагмент самодостаточного кода, т.е. для его функционирования не требуется наличия дополнительных библиотек;
- является самоустановливаемым модулем, который может быть включен в состав системы, исключен из ее состава или заменен на другой модуль, например, принадлежащий другому производителю, при минимальном участии пользователя;
- может повторно использоваться в различных контекста;
- при работе с ПК используются механизмы динамического связывания;
- может быть объединено с другими ПК в целях создания более крупных ПК;
- пользователи применяют ПК преимущественно по принципу черного ящика, т.е. пользователю известны только интерфейсы, но не внутренняя структура системы.

ПК может быть независимо поставлен или не поставлен, добавлен в состав некоторой системы или удален из нее, в том числе, может включаться в состав систем других поставщиков.

ПК можно определить как фрагмент самодостаточного, самоустанавливаемого кода, обладающий хорошо определенной функциональностью и имеющий четко определенные интерфейсы. ПК могут агрегатироваться с другими компонентами в более сложные системы. Компоненты и системы компонентов могут повторно использоваться в различных контекстах. Пользователи используют компоненты преимущественно по принципу черного ящика, т.е. пользователю известны только интерфейсы, но не внутренняя структура ПК.

На основе вышеизложенного можно говорить о следующих основных различиях между компонентами и объектами:

- как ПК, так и объекты ориентированы на повторное использование кода, однако объекты ориентированы преимущественно на повторное использование на низком уровне, а ПК — на высокоуровневое повторное использование кода;
- ПК в большей степени ориентированы на интерфейсы, чем объекты;

• ПК разрабатываются в рамках конкретного фреймворка, а объекты — в рамках конкретного языка программирования;

• объекты тесно связаны с конкретным языком программирования в то время как ПК в явном виде не связаны с конкретным языком программирования, но связаны с платформой, а платформа может быть связана с конкретным языком, например, JEE связана с Java.

Обычно объекты не являются автономными модулями, которые можно легко заменить на другие объекты, поскольку они, как правило, имеют вложенные объекты. Для выполнения процедуры замены объекта обычно требуется перекомпилировать приложение, а для замены компонента часто оказывается достаточным поместить файл, в котором находится исполняемый код, в заданное место и отредактировать конфигурационный файл [59].

Для функционирования ПК, как правило, необходимо наличие соответствующей инфраструктуры, которая позволяет компонентам находить друг друга и взаимодействовать по определенным правилам. Набор правил, определяющих интерфейсы ПК и их реализаций, а также правил, по которым ПК работают в системе и взаимодействуют друг с другом, называют ***компонентной моделью*** (component model) [45]. В компонентную модель входят также правила, регламентирующие жизненный цикл ПК. Взаимодействовать друг с другом могут только ПК, построенные в рамках одной модели.

Для работы ПК необходим некоторый набор базовых служб (basic services), которые обеспечивают, например, нахождение компонентов в распределенной среде, обеспечение обмена данными через сеть.

Набор таких базовых, необходимых для функционирования большинства компонентов служб, вместе с поддерживаемой с их помощью компонентной моделью называется ***компонентной средой*** (компонентным фрейворком, component framework).

Компонентные технологии можно рассматривать как одну из фаз развития технологий разработки распределенных систем, при этом можно выделить следующие основные фазы (рис. 4.1):

• сокеты;
• вызов удаленных процедур;
• системы распределенных объектов;
• компонентные технологии;
• сервисно-ориентированные системы.

Перечисленные технологии появлялись именно в таком порядке. Переход к следующей фазе можно рассматривать как достижение некоторого уровня зрелости, поскольку технологии нижележащего уровня используются в качестве сервисов более низкого уровня. В частности, вызов удаленных процедур основывается на использовании сокетов. Системы распределенных объектов, в свою очередь, базируются на вызове удаленных процедур, компонентные технологии могут использовать такие механизмы, как RMI, а Web-сервисы могут быть реализованы как компоненты.

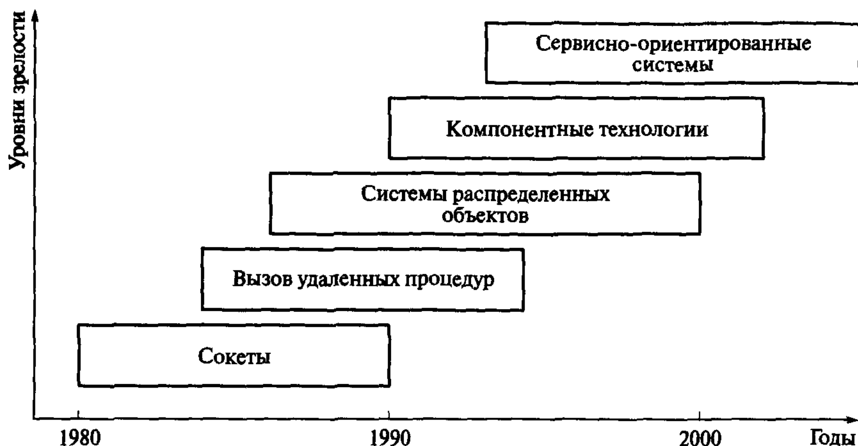


Рис. 4.1. Фазы развития технологий разработки распределенных систем

Известно достаточно много разных компонентных технологий. Некоторые из них остались на уровне теоретических исследований, однако ряд технологий активно используются на практике в течение уже многих лет. К последней группе можно отнести такие компонентно-ориентированные (component based) технологии, как JavaBeans, EJB, CORBA, ActiveX, VBA, COM, DCOM, .Net компоненты. Принципиально мультиагентные технологии также можно рассматривать как разновидность компонентных технологий.

4.2. Квазикомпонентно-ориентированные технологии

Рассмотренные ранее технологии не являются компонентно-ориентированными, но лежат в их основе.

Сокеты. Самым старым способом взаимодействия между элементами распределенных приложений являются сокеты, которые позволяют связывать приложения напрямую, записывая данные и читая данные из сокета. API для сокетов является низкоуровневым и предоставляет максимальный контроль над передаваемыми данными.

Однако сокеты плохо подходят для работы со сложными структурами данных, особенно если это программные компоненты.

Работа с сокетами в самом общем виде выглядит следующим образом. Любой процесс может создать серверный сокет и привязать его к какому-нибудь порту. После этого сервер переходит в режим ожидания и ожидает запрос со стороны клиента. Клиент также создает сокет, через который он может взаимодействовать с сервером. Взаимодействие может реализовываться как с установлением соеди-

нения, так и без установления соединения. В первом случае в качестве транспорта используется протокол TPC, а во втором — UDP.

Механизм работы с сокетами поддерживается практически во всех современных языках программирования. Самую подробную информацию по программированию сокетов можно найти в [16].

Вызов удаленных процедур. Основная идея механизма сокетов состоит в использовании операций send и receive для обмена данными между процессами, которые выполняются на разных хостах. Основной недостаток сокетов состоит в сложности программирования, поскольку программирование осуществляется на низком уровне.

Идея вызова удаленных процедур состоит в обеспечении возможности программ вызывать процедуры, находящиеся на других машинах. Когда процесс, запущенный на хосте A, вызывает процедуру с хоста B, процесс на хосте A приостанавливается, а выполнение вызванной процедуры происходит на хосте B. Информация может быть передана от вызывающего процесса к вызываемой процедуре через параметры и возвращена процессу в виде результата выполнения процедуры. Для программиста вызов удаленной процедуры ничем не отличается от вызова локальной процедуры. Данный метод известен под названием удаленный вызов процедур (Remote Procedure Call, RPC).

Использование подпрограмм (процедур) в программе — традиционный способ структурировать задачу и сделать ее более ясной. Обычно подпрограммы собираются в библиотеки. Применительно к локальным вызовам данный подход используется давно и повсеместно.

Удаленный вызов процедуры существенным образом отличается от традиционного локального с позиции реализации, однако с точки зрения программиста такие отличия практически отсутствуют.

В случае локального вызова программа передает параметры в вызываемую процедуру и получает результат работы через стек. В случае удаленного вызова передача параметров превращается в передачу запроса по сети, а результат выполнения процедуры находится в пришедшем ответе. Если основная программа и реализуемая подпрограмма находятся на хостах с разной архитектурой, работающих под управлением разных операционных систем, и написаны на разных языках программирования, то задача корректной реализации удаленного вызова процедур усложняется.

Первая проблема состоит в том, что нельзя напрямую передать управление программе, которая работает в другом адресном пространстве. Это можно сделать только с помощью заглушек. Заглушка (stub), или заместитель (proxy), представляет собой фрагмент кода, который несет ответственность за обмен данными. В самом общем виде ситуация выглядит следующим образом.

Когда необходимо вызвать удаленную процедуру на вызывающей стороне, основная программа обращается к клиентской заглушке,

которая связывается с серверной заглушкой и передает ей параметры вызова. Серверная заглушка получает параметры вызова и уже от своего имени вызывает требуемую подпрограмму. Результаты выполнения передаются серверной заглушке. Получив результаты, серверная заглушка передает их клиентской заглушке, которая, в свою очередь, возвращает их основной программе.

Рассмотрим данный процесс более подробно.

При описании механизма RPC будем традиционно называть вызывающий процесс клиентом, а удаленный процесс — процессом, реализующим процедуру, или сервером.

Можно выделить следующие шаги вызова удаленной процедуры.

1. Программа клиент выполняет локальный вызов процедуры посредством обращения к заглушке на своей стороне.

2. Заглушка, расположенная на стороне сервера, перекодирует запрос в стандартный формат. Этот процесс называют маршаллингом (marshalling).

3. Заглушка, расположенная на стороне клиента, связывается с серверной заглушкой.

Задача связывания распадается на две подзадачи:

- нахождение удаленного хоста с требуемым сервером;
- нахождение требуемого серверного процесса на данном хосте.

Обычно эти задачи решаются посредством использования централизованного репозитария, в котором хранится информация об имеющихся сервисах. Данный репозитарий располагается по адресу, который известен всем серверам и клиентам. При запуске сервера он регистрирует свои сервисы в репозитарии, а клиент, когда ему требуется выполнить связывание, обращается к репозитарию и считывает информацию о месте нахождения сервиса.

После того как клиент узнал местонахождение сервера, клиентская заглушка связывается через сокет с серверной заглушкой и пересылает ей параметры вызова, после чего обычно переходит в режим ожидания.

4. Получив запрос на выполнение удаленной процедуры, серверная заглушка выполняет операцию демаршаллинга (unmarshalling). Демаршаллинг предполагает выполнение операции обратной операции маршаллинга, т.е. преобразование параметров запроса из сетевого во внутренний формат.

5. Серверная заглушка вызывает требуемую процедуру.

6. Процедура выполняется на сервере, результаты выполнения процедуры передаются серверной заглушке.

7. Серверная заглушка выполняет операцию маршаллинга и отправляет результат клиентской заглушке.

8. Получив результат выполнения удаленной процедуры, клиентская заглушка выполняет операцию демаршаллинга и передает результат основной программе.

При работе с удаленными процедурами параметры передаются по значению. Передача параметров по ссылке не используется.

Работа с RPC предполагает использование соответствующей среды разработки, которая значительно облегчает процесс программирования. Ключевым элементом является генератор заглушек.

Генератор заглушек функционирует следующим образом. На специальном языке описания интерфейсов описываются параметры вызова. Это очень простой непроцедурный язык, который позволяет определить параметры аргументов и результатов.

Описание интерфейса, представляющее собой обычный текстовый файл, обрабатывается с помощью генератора заглушек. На выходе получаем текстовые файлы, в которых содержится код клиентской и серверной заглушек. Обычно это код на языке С. Сгенерированный код вставляется в программы клиента и сервера, и выполняется их компиляция. Если клиент и сервер работают на разных платформах, то процесс генерации выполняется как на серверной, так и на клиентской платформах.

Использование RPC существенно облегчает работу прикладного программиста, поскольку ему не требуется иметь дело с сокетами.

Приведенное выше описание RPC достаточно поверхностное. Подробное описание можно найти, например в [16].

Идея RPC была предложена и реализована еще в 1980-х гг. и достаточно долго и успешно использовалась.

Использование механизма работы с удаленными процедурами имеет существенные недостатки:

- RPC ориентирован на процедурный стиль программирования, а подавляющее большинство современных приложений — объектно-ориентированные;
- в рамках RPC реализуются статические вызовы, т.е. заглушки встраиваются в текст как клиента, так и сервера, на этапе разработки.

Среда распределенных вычислений (Distributed Computing Environment, DCE). Механизм удаленных вызовов процедуры был тщательно адаптирован для использования в качестве основы систем промежуточного уровня и вообще распределенных систем и достаточно хорошо работал при использовании таких языков программирования как С. Дальнейшим развитием данного подхода можно считать Среду распределенных вычислений (Distributed Computing Environment, DCE), разработанную организацией OSF (Open Software Foundation), которая позже была переименована в Open Group. DCE изначально она была разработана под UNIX, однако позже был были созданы версии для других ОС.

DCE включает следующие основные сервисы: службу распределенных файлов, службу каталогов, службу распределенного времени, кроме того, имеются поддержка работы с нитями (Threads),

вызов удаленных процедур и аутентификация, служба безопасности и файловый сервис.

Служба распределенных файлов (*distributed file service*) представляет собой всемирную файловую систему, предоставляющую прозрачные методы доступа к любому файлу системы одинаковым образом.

Служба каталогов (*directory service*) используется для отслеживания местонахождения любого из ресурсов системы. Служба каталогов позволяет процессу запрашивать ресурсы, не зная, где они находятся, если это необходимо для процесса.

Служба безопасности (*security sewice*) позволяет защищать ресурсы любого типа, кроме того, получение некоторых данных может быть открыто только тем, кому это разрешено.

Служба распределенного времени (*distributed time service*) предоставляет механизмы синхронизации часов различных хостов.

Модель программирования, лежащая в основе всей системы DCE, — это модель клиент-сервер, в которой связь между клиентами и серверами осуществляется посредством использования RPC. В основе своей DCE представляет собой традиционную, основанную на RPC, систему. Время появления DCE совпало с появлением объектно-ориентированных языков и поэтому разработчики были вынуждены ввести поддержку работы с объектами, однако эта поддержка носит достаточно ограниченный характер. Правильным будет считать, что DCE представляет собой «мост» между RPC и системами распределенных объектов.

Распределенные объекты были добавлены в DCE в форме расширений языка определения интерфейсов (IDL) вместе с привязкой к языку C++. Другими словами, распределенные объекты в DCE описываются на IDL и реализуются на C++.

Распределенные объекты представляют собой удаленные объекты, реализация которых находится на сервере.

Сервер создает объекты C++ и обеспечивает доступ к удаленным клиентам.

Других способов создания распределенных объектов не существует.

В рамках DCE поддерживаются два типа распределенных объектов:

- динамические распределенные объекты (distributed dynamic objects);
- именованные распределенные объекты (distributed named objects).

Динамические распределенные объекты создаются сервером по требованию клиента, к ним имеет доступ только один клиент.

Именованные распределенные объекты создаются сервером для совместного использования несколькими клиентами и регистрируются службой каталогов, так что клиент может найти объект и вы-

полнить привязку к нему. За объектом сохраняется уникальный идентификатор. Все обращения к удаленным объектам в DCE производятся средствами RPC. Клиент, обращаясь к методу, передает серверу идентификатор объекта, идентификатор интерфейса, содержащего метод, идентификацию самого метода и параметры.

Сервер поддерживает таблицу объектов для идентификации объекта, к которому обратился клиент, затем он выбирает запрошенный метод и передает ему параметры.

DCE предоставляет возможность помещать объекты при необходимости во вспомогательное временное хранилище данных.

У распределенных объектов в DCE имеется серьезная проблема, связанная с их чрезвычайной близостью с RPC, которая состоит в том, что не существует механизма прозрачных ссылок на объекты.

Клиент может использовать только дескриптор привязки (binding handle), ассоциированный с именованным объектом, который может быть преобразован в строку и в таком виде передан другому процессу. Более подробную информацию по DCE можно найти в [17].

Программный интерфейс вызова удаленных методов в Java (Java Remote Method Invocation, RMI). В DCE распределенные объекты были добавлены достаточно искусственно в качестве расширения вызовов удаленных процедур. При этом клиент работает со ссылкой на удаленную процедуру для объекта. Фактически ссылки на объекты отсутствуют.

В качестве полноценной системы работы с распределенными объектами можно рассматривать Java RMI. В Java распределенные объекты интегрированы с языком.

Java поддерживает распределенные объекты в форме удаленных объектов, т. е. объектов, тело которых постоянно находится на одном и том же хосте, а интерфейсы доступны удаленным процессам. Интерфейсы реализованы обычным образом через заглушки, которые предоставляют интерфейсы, идентичные интерфейсам удаленных объектов. При этом заглушка для клиента имеет вид локального объекта, находящегося в адресном пространстве клиента.

В Java между локальными и удаленными объектами существует ряд различий. Во-первых, различаются механизмы клонирования локальных и удаленных объектов. При клонировании локального объекта создается новый объект такого же типа, что и исходный с таким же состоянием. При клонировании удаленного объекта операция клонирования выполняется только на сервере и приводит к созданию точной копии объекта в адресном пространстве сервера. Заместители объекта не клонируются. Поэтому, если клиент на удаленной машине хочет получить доступ к новому объекта на сервере, то необходимо выполнить повторную привязку.

Второе различие между локальными и удаленными объектами в Java заключается в семантике блокировки объектов. Для блокиров-

ки объекта в Java достаточно объявить один из методов синхронизируемым (synchronized). Если два процесса одновременно вызовут этот метод, то доступ к нему получит только один. Таким образом, можно гарантировать, что доступ к внутренним данным объекта реализуется только последовательно. Java RMI ограничивает блокировку удаленных объектов блокировкой заместителей, т.е. если процессы используют разные заместители, то удаленные объекты невозможно защитить от одновременного доступа процессов.

Java в ходе обращений к удаленным методам скрывает большую часть различий между ними. В частности, при работе с RMI в качестве параметров можно передавать любой простой или объектный тип. В терминологии Java это означает, что типы сериализуемы (serializable). Сериализации можно подвергнуть большинство объектов, не зависящих от платформы, — таких как дескрипторы файлов или сокеты.

Основное различие между удаленными и локальными объектами состоит в том, что локальные объекты передаются по значению, а удаленные — по ссылке. В Java ссылка на удаленный объект содержит сетевой адрес и конечную точку сервера, а также локальный идентификатор необходимого объекта в адресном пространстве сервера.

Удаленный объект состоит из двух различных классов:

• класса сервера, содержащего реализацию кода сервера;

• класса клиента, содержащего реализацию кода клиента. Класс клиента содержит реализацию заместителя.

Таким образом, заместитель обладает всей информацией, необходимой для обращения клиента к методу удаленного объекта. В Java заместители можно сериализовать, подвергнуть маршалингу и переслать в виде набора байтов другому процессу, в котором он может быть подвергнут обратной операции (демаршалингу) и использован для обращения к методам удаленного объекта. Косвенным результатом этого является тот факт, что заместитель может быть использован в качестве ссылки на удаленный объект.

При маршалинге заместителя вся его реализация, т.е. его состояние и код, превращаются в последовательность байтов. Маршалинг подобного кода мало эффективен и может привести к слишком объемным ссылкам. Поэтому при маршалинге заместителя в Java на самом деле происходит генерация дескриптора реализации, точно определяющего, какие именно классы необходимы для создания заместителя. Возможно некоторые из этих классов придется предварительно загрузить с удаленного узла. Дескриптор реализации в качестве части ссылки на удаленный объект заменяет передаваемый при маршалинге код. В результате ссылки на удаленные объекты в Java имеют размер порядка нескольких сотен байт.

Такой подход к ссылкам на удаленные объекты отличается высокой гибкостью и представляет собой одну из отличительных особенностей RMI в Java. В частности, это позволяет оптимизировать

решение под конкретный объект. Так, рассмотрим удаленный объект, состояние которого изменяется только один раз. Мы можем превратить этот объект в настоящий распределенный объект путем копирования в процессе привязки всего его состояния на клиентскую машину. Каждый раз при обращении клиента к методу он работает с локальной копией. Чтобы гарантировать согласованность данных, каждое обращение проверяет, не изменилось ли состояние объекта на сервере, и при необходимости обновляет локальную копию. Таким же образом методы, изменяющие состояние объекта, передаются на сервер. Разработчик удаленного объекта должен разработать только код, необходимый для клиента, и сделать его динамически подгружаемым при присоединении клиента к объекту.

Возможность передавать заместителя в виде параметра существует только в том случае, если все процессы работают под управлением одной и той же виртуальной машины. Другими словами, каждый процесс работает в одной и той же среде исполнения. Переданный при маршалинге заместитель просто подвергается демаршалингу на приемной стороне, после чего полученный код заместителя можно выполнять. В противоположность этому в DCE, например, передача заглушек невозможна, поскольку разные процессы могут запускаться в средах исполнения, различающихся языком программирования, операционной системой и аппаратным обеспечением. Вместо этого в DCE производится компоновка (динамическая) с локальной заглушкой, скомпилированной в расчете на среду исполнения конкретного процесса. Путем передачи ссылки на заглушку в виде параметра RPC достигается возможность выхода за границы процесса.

Для того чтобы клиент мог найти удаленный объект, на сервере имеется RMI-реестр (RMI registry), представляющий собой небольшую базу данных, в которой хранится информация об удаленных объектах, находящихся на этом сервере. Каждый удаленный объект должен быть зарегистрирован в RMI-реестре. RMI-реестр обслуживает процесс демон, который по умолчанию прослушивает порт 1099 и ждет запросы от клиента. RMI-реестр рассматривается как удаленный объект. Кроме того, на сервере создается и хранилище заглушек.

RMI-запрос выполняется в следующей последовательности:

1. Удаленный объект создается на сервере, регистрируется в RMI-реестре и ждет запрос от клиента, прослушивает порт 1099.

2. RMI-реестр прослушивает порт, по умолчанию 1099, и ждет клиента.

3. Клиент, который хочет обратиться к удаленному объекту обращается к RMI-реестру и запрашивает объект по имени.

4. Реестр отыскивает заместителя объекта (stub) и пересылает ее клиенту. В заместителе, в частности, содержится информация о местоположении.

5. Заместитель запускается и запрашивает у сервера описание методов удаленного объекта, указав его местоположение.

6. Сервер составляет список сигнатур методов удаленного объекта и отправляет их заместителю.

7. Клиент обращается к методам заместителя так, как будто это собственные методы.

8. Если аргументы метода являются локальные объекты (находятся на стороне клиента), то заместитель выполняет сериализацию и отправляет их на сервер. Если аргументы находятся на сервере, то серверу пересылается ссылка на них.

9. Сервер выполняет десериализацию параметров, передает их методу удаленного объекта, который выполняется. Результат передается заместителю, находящемуся на серверной стороне, который выполняет их сериализацию и отсылает клиенту.

RMI может работать не только с реестром RMI, но и использовать службы каталогов, включая Java Naming and Directory Interface (JNDI).

4.3. Технологии, основанные на объектной модели компонентов COM+, .NET

4.3.1. Объектная модель компонентов (COM)

В самом общем виде идея COM состоит в том, что одна часть ПО должна получать доступ к сервисам, предоставляемым другой частью, причем используется доступ ко всем видам программных сервисов независимо от способа их реализации. COM применяет стандартный механизм.

В COM любая часть программного обеспечения реализует свои сервисы как один или несколько объектов COM. Каждый такой объект поддерживает один или несколько интерфейсов, состоящих из методов. Каждый метод — это процедура или функция, которая выполняет требуемое действие и может быть вызвана программным обеспечением, использующим данный объект (клиентом объекта).

Клиенты получают доступ к сервисам объекта COM только через вызовы методов интерфейсов объекта — у них нет непосредственного доступа к данным объекта.

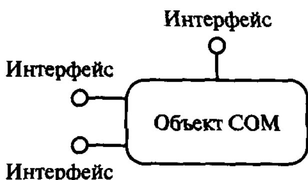


Рис. 4.2. Пример объекта COM

Объект COM может поддерживать более одного интерфейса, например, объект COM, показанный на рис. 4.2, имеет три интерфейса.

Каждый интерфейс включает множество методов. В число этих методов входят три стандартных метода, которые будут рассмотрены далее, и произвольное чис-

ло методов, определяемых пользователем. Однажды определенный интерфейс нельзя изменять и дополнять. Если появляется необходимость изменить интерфейс, то создается новый интерфейс.

Рассмотрим пример. Допустим разработчиками был создан объект COM, для реализации проверки правописания, единственный интерфейс ISpeller, который кроме стандартных содержит три метода: FindWord (); AddWord (), RemoveWord (), которые позволяют находить слово в словаре, добавлять слово в словарь и удалять слово из словаря соответственно.

Если позднее появляется идея расширить функциональные возможности объекта COM за счет включения функции поиска синонимов, то поскольку интерфейс нельзя изменять, создают новый интерфейс, который можно назвать, например IThesaur. Помимо трех стандартных методов он содержит единственный метод GetSynonym ().

Интерфейсы COM. Каждый поддерживаемый объектом интерфейс, по сути, — контракт между этим объектом и его клиентами. Объект должен поддерживать методы интерфейса в соответствии со спецификацией, а клиент обязуется корректно вызывать методы.

Интерфейс COM также включает в себя набор функций, которые реализуются компонентами и используются клиентами. Но COM дает более точное определение интерфейса. В COM интерфейсом является определенная структура в памяти, содержащая массив указателей на функции. Каждый элемент массива содержит адрес функции, реализуемой компонентом. Для клиента компонент представляет собой набор интерфейсов. Клиент может взаимодействовать с компонентом COM только через интерфейс.

У каждого интерфейса COM имеется два имени. Одно из них предназначено для использования человеком, а второе — для использования ПО.

Первое имя представляет собой строку символов, например ISpeller. По соглашению читабельные имена большинства COM-интерфейсов начинаются с буквы I (от interface).

Второе имя — машинное имя, которое представляет собой идентификатор интерфейса IID, имеющий длиной 128 бит (16 байт), который называют также GUID (Globally Unique IDentifier — глобально уникальный идентификатор).

Этот идентификатор уникален во времени и в пространстве. Идентификатор можно сформировань самостоятельно с помощью утилиты. Он состоит из двух частей. Первая часть (48 бит) — это обычно номер сетевой карты хоста, на котором работает утилита, а вторая часть — это текущее значение системых часов. Если в машине отсутствует сетевая карта, то MAC-адрес заменяется случайным числом.

Каждый объект COM должен поддерживать интерфейс IUnknown, иначе он не будет объектом COM. Интерфейс IUnknown включает три метода: QueryInterface, AddRef и Release. Все интерфейсы наследуют от IUnknown, его методы можно вызывать через любой из указате-

лей на интерфейс. Обычно первый указатель на интерфейс объекта клиент получает при создании объекта. Затем клиент может получить ссылку на требуемый интерфейс, запросив у объекта указатели с помощью IUnknown::QueryInterface.

Поскольку все интерфейсы COM наследуют IUnknown, в каждом интерфейсе есть функции QueryInterface, AddRef и Release. Внутри COM объекта имеется виртуальная таблица (рис. 4.3), которая содержит указатели на реализуемые методы. Первые три строчки — это указатели на QueryInterface, AddRef и Release.

При помощи QueryInterface клиент определяет, поддерживается ли тот или иной интерфейс. Метод QueryInterface использует в качестве аргумента (Interface Identifier — IID) GUID и возвращает указатель на определенный интерфейс.

Пара функций AddRef и Release реализуют технику управления памятью, основанную на подсчете ссылок (reference counting). Подсчет ссылок — простой и быстрый способ, позволяющий управлять жизненным циклом компонента. Компонент COM поддерживает счетчик ссылок. Механизм очень прост. Когда клиент получает в свое распоряжение некоторый интерфейс, значение этого счетчика увеличивается на единицу. Когда клиент заканчивает работу с интерфейсом, значение на единицу уменьшается. Когда оно доходит до нуля, компонент удаляется из памяти. Клиент также увеличивает счетчик ссылок, когда создает новую ссылку на уже имеющийся у него интерфейс. Внутри функции незачем подсчитывать ссылки для указателей на интерфейсы, хранящиеся в локальных переменных. Однако подсчет ссылок необходим при всяком копировании указателя

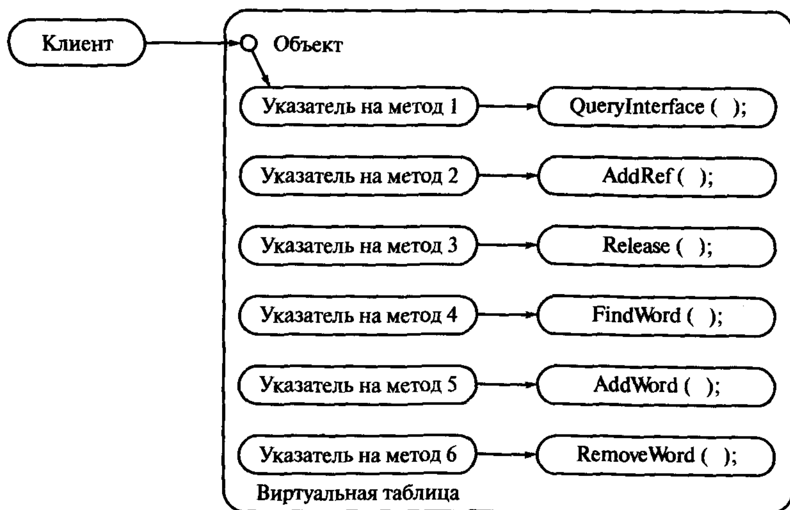


Рис. 4.3. Виртуальная таблица объекта COM

в глобальную переменную или из нее: глобальная переменная может освободиться в любой момент и в любой функции. С точки зрения клиентов подсчет ссылок осуществляется для интерфейсов, а не для компонентов, для реализации компонента это не имеет значения.

Компонент может поддерживать отдельные счетчики для каждого из интерфейсов, а может иметь один общий счетчик. Реализация не имеет значения до тех пор, пока клиент убежден, что подсчет ссылок ведется для самих интерфейсов. Поскольку компонент может реализовывать подсчет для каждого интерфейса, клиент не должен предполагать обратного. Подсчет ссылок для каждого интерфейса в отдельности означает, что клиент должен вызывать AddRef именно для того указателя, с которым собирается работать, а не для какого-нибудь другого. Клиент также должен вызывать Release именно для того указателя, с которым закончил работу. Если клиент не вызвал функцию Release для некоторого из интерфейсов компонента, то компонент будет продолжать занимать память. Еще более неприятная ситуация возникает в том случае, когда клиент дважды вызовет функцию Release, — в этом случае объект будет уничтожен досрочно.

Допустимо существование в любой данный момент времени одного, двух или многих активных объектов одного класса.

Серверы объектов COM. Библиотека COM. Каждый объект COM реализуется внутри некоторого сервера.

Один сервер может поддерживать несколько классов. Выделяются следующие типы серверов:

- сервер «в процессе», который реализуется в виде динамической библиотеки (.dll), т.е. исполняется в одном адресном пространстве с клиентом;
- локальный сервер, в котором объекты реализованы в отдельном процессе (.exe файл), исполняющемся на том же хосте, что и клиент;
- удаленный сервер, в котором объекты реализованы в DLL или в отдельном процессе. Объекты расположены на удаленном по отношению к клиенту хосте (в этом случае используется распределенная COM (DCOM)). С точки зрения клиента, объекты, реализованные любой из перечисленных выше трех разновидностей серверов, выглядят одинаково.

Доступ к методам объектов во всех случаях осуществляется через указатели интерфейсов.

Клиент может получить указатель на один из интерфейсов разными способами. Например, указатель может быть передан другим клиентом, либо клиент может получить его от моникера (моникеры будут рассмотрены далее).

В любой системе, поддерживающей COM, должна иметься реализация библиотеки COM, которая предоставляет клиентам механизм запуска серверов объектов. Доступ к сервисам библиотеки COM осу-

ществляется через вызовы обычных функций, а не методов интерфейсов COM-объектов. Обычно имена функций библиотеки COM начинаются с «Co» — например, CoCreateInstance.

Создание объектов COM. Самый простой способ создания одного неинициализированного экземпляра объекта показан на рис. 4.4.

Можно выделить четыре основные фазы создания объекта COM:

1) клиент вызывает функцию библиотеки COM CoCreateInstance;

2) библиотека COM находит в системном реестре запись, соответствующую классу данного объекта;

3) библиотека COM запускает сервер COM и возвращает клиенту указатель на требуемый интерфейс;

4) клиент вызывает требуемый метод.

При вызове функции CoCreateInstance в качестве аргументов библиотеке COM передаются CLSID и IID первого интерфейса, указатель которого необходим клиенту. Параметры CoCreateInstance позволяют также клиенту указать, какой тип сервера должен быть запущен библиотекой COM.

Получив запрос, библиотека COM, точнее Диспетчер управления сервисами, или SCM (Service Control Manager), просматривает записи в реестре, которые содержат информацию о местоположении сервера, способного создать экземпляр класса объекта. После того как сервер найден, SCM запускает его, если сервер еще не запущен. Запущенный сервер создает экземпляр класса объекта и возвращает указатель на запрошенный интерфейс библиотеке COM. Последняя, в свою очередь, передает данный указатель клиенту, который теперь может выполнять вызовы методов этого интерфейса. Так как результатом данного процесса является создание неинициализированного объекта, то клиент обычно запрашивает интерфейс, через который объект может быть инициализирован, хотя это и не обязательно.

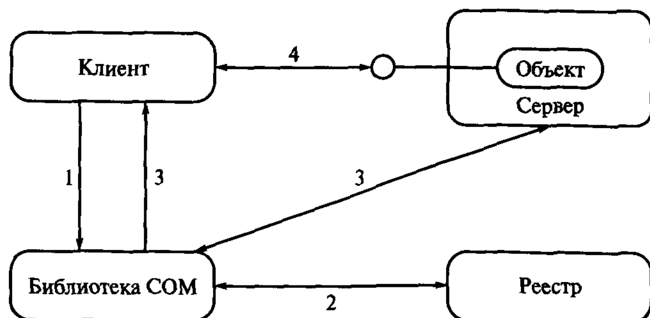


Рис. 4.4. Способ создания одного неинициализированного экземпляра объекта

Поддержка удаленных объектов обеспечивается средствами DCOM. Процесс во многом аналогичен созданию локального объекта: клиент выполняет тот же вызов библиотеки COM, SCM просматривает системный реестр и т.д. Если указан удаленный сервер, COM установит для создания экземпляра объекта связь с удаленных хостом. Данный запрос выполняется вызовом удаленной процедуры. Просмотрев свой реестр, удаленная система находит исполняемый файл сервера и создает экземпляр объекта. Так же, как и в случае локального сервера, указатель возвращается на интерфейс, после чего клиент может вызывать методы вновь созданного объекта. Для клиента запуск объекта выполняется одинаково независимо от того, каким сервером реализован объект: «в процессе», локальным или удаленным; данное различие должно учитываться клиентом, только тогда, когда он сам считает это необходимым.

Если нужно создать только один объект, то проще всего создать его с помощью CoCreatelnstance.

В случае, если требуется создать много экземпляров объектов одного и того же класса, то можно использовать фабрику класса (class factory) — объект, способный создавать другие объекты. Каждая фабрика класса предназначена для создания объекта одного конкретного класса.

Фабрика класса — это объект COM, доступ к которому осуществляется через интерфейсы, поддерживающие IUnknown и т.д. И все же они необычные объекты, так как могут создавать другие объекты COM.

Даже когда клиент просто вызывает CoCreatelnstance, реализация этой функции в библиотеке COM создает объект с помощью фабрики класса. CoCreatelnstance скрывает эти детали от клиента, но на самом деле используют методы интерфейса IClassFactory, описываемые далее.

Для того чтобы COM-объект являлся фабрикой класса, он должен поддерживать интерфейс IClassFactory, который содержит два метода: CoCreatelnstance и LockServer.

Метод CoCreatelnstance создает новый экземпляр класса, объекты которого может создавать данная фабрика. Клиент не передает этому методу в качестве параметра CLSID, так как класс объекта неявно определяется самой фабрикой. И все же клиент задает IID, чтобы получить указатель на нужный ему интерфейс.

Метод LockServer позволяет клиенту сохранить сервер загруженным в память. Объект-фабрика, как и другие объекты, поддерживает собственный счетчик ссылок, для учета количества использующих его клиентов. Однако по разным (очень сложным) соображениям этого счетчика недостаточно, чтобы удерживать сервер загруженным в память. Чтобы сервер гарантированно продолжал работать, можно использовать IClassFactory::LockServer.

В некоторых случаях интерфейс IClassFactory слишком прост. На сегодня имеется новый интерфейс IClassFactory2, добавляющий

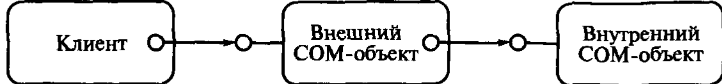


Рис. 4.5. Использование внешним объектом внутреннего объекта в качестве сервера

новые возможности, в частности, он поддерживает и еще несколько методов, связанных с лицензированием.

Описанный выше процесс создания COM-объектов создает некий абстрактный экземпляр данного класса. Для того чтобы завершить создание экземпляра объекта, необходимо в общем случае загрузить его данные.

Повторное применение COM-объектов. Объектно-ориентированные технологии в качестве основного механизма повторного использования существующего кода применяют, как правило, наследование реализации, когда новый объект наследует реализацию методов существующего объекта.

Применительно к компонентным технологиям наследование реализации не является лучшим решением, поскольку при таком подходе бывает сложно создавать автономные модули, с которыми можно выходить на рынок.

В рамках COM в качестве базовых механизмов повторного применения выступают включение (containment) и агрегирование (aggregation), в основе которых лежит некоторая взаимосвязь объектов. В терминологии COM «внешним» (outer) называется объект, использующий сервисы «внутреннего» (inner). Внешним объектом выступает клиент внутреннего, либо их взаимосвязь может быть несколько более тесной.

Основная идея включения состоит в том, что внешний объект использует внутренний объект в качестве сервера (рис. 4.5), при этом к методам внутреннего объекта обращаться может только внешний объект.

Благодаря своей простоте, включение является широко распространенным механизмом повторного применения в COM.

Агрегирование предполагает возможность внешнему объекту представлять в качестве собственных интерфейсы, которые на самом деле реализуются внутренним объектом (рис. 4.6). Когда клиент запраши-

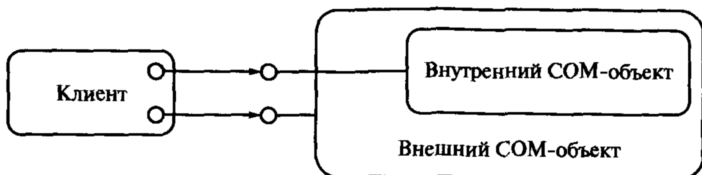


Рис. 4.6. Добавление методов внутреннего объекта к методам внешнего объекта

вает у внешнего объекта указатель на такой интерфейс, этот объект возвращает указатель на интерфейс внутреннего, агрегированного объекта. Методы внутреннего объекта добавляются (агрегируются) к методам внешнего объекта. Клиент ничего об этом не знает: возвращенный интерфейс обеспечивается для него только одним известным ему объектом внешним.

Перманентность данных. Большинство приложений требует сохранять данные. Данные приложения могут сохраняться разными способами, например, в БД, в файлах или хранилищах данных.

Применительно к компонентам, когда речь идет о способах сохранения и восстановления их состояния используют термин «перманентные» (persistent). Перманентность данных позволяет объекту прекращать свое исполнение, а когда повторно появляется необходимость продолжить работу с компонентом, то экземпляр объекта будет создан снова и продолжает работу с того же места, просто загрузив перманентные данные.

Механизм, с помощью которого объекты сохраняют и загружают свои перманентные данные, иногда называют сервисом перманентности (persistent service).

Можно выделить два альтернативных подхода к управлению перманентностью. В первом случае компонент автономно определяет, когда делать свои данные перманентными. Во втором случае перманентностью в явном виде управляет пользователь, указывая объекту, когда загружать и сохранять свои данные.

COM-объекты предоставляют два разных набора интерфейсов для поддержки перманентности. Первый дает возможность COM-объектам работать со структурированным хранилищем (Structured Storage). Второй набор стандартных интерфейсов позволяет клиенту управлять перманентностью COM-объекта. COM-объект может поддерживать один или несколько таких интерфейсов, объединенных под общим названием интерфейсы IPersist.

Хранение состояния каждого отдельного компонента в собственном файле не является лучшим решением, особенно, если имеется много компонентов. Предпочтительным является использование БД или создание хранилища.

Структурированное хранилище используется для хранения перманентных данных нескольких компонентов в общем файле. Обобщенная структура хранилища показана на рис. 4.7.

Файл структурированного хранилища состоит из хранилищ (storage), которые выполняют функции каталогов, и потоков (stream), выполняющих функции файлов. Каждый составной файл содержит корневое хранилище, в котором могут располагаться другие потоки и хранилища. В этих хранилищах, в свою очередь, могут быть дополнительные потоки, хранилища и т.д. Это очень похоже на обычную файловую систему за исключением того, что вся конструкция помещена в один файл.

Рис. 4.7. Обобщенная структура хранилища

Хранилища и потоки рассматриваются как COM-объекты, и доступ к каждому из них осуществляется через соответствующий интерфейс. Хранилища поддерживают интерфейс IStorage, а потоки — интерфейс IStream.

Моникеры. При работе с компонентами крайне полезно иметь способ идентификации конкретного экземпляра объекта.

В общем случае для обращения к конкретному экземпляру компонента необходимо указать два элемента: методы объекта и его перманентные данные. Для объектов, не имеющих перманентных данных, достаточно указать методы. Но для большинства объектов необходимо и то и другое.

Сам по себе COM не предлагает способа именования экемпляра объекта. Клиент может создать абстрактный экземпляр компонента, вызвав CoCreateInstance и передав ей сответствующий CLSID. Если у компонента имеются перманентные данные, то клиент может использовать, например один из интерфейсов IPersist.

Для этого клиент должен знать CLSID объекта и способ найти место хранения перманентных данных объекта. Для решения этой задачи COM предлагает использовать моникер.

Моникер (moniker) — жаргонный эквивалент слова nickname (уменьшительное имя, прозвище).

Моникер представляет собой COM-объект, который поддерживает интерфейс — IMoniker. Каждый моникер имеет собственные перманентные данные, в составе которых все, что необходимо моникеру для запуска и инициализации одного экземпляра объекта, идентифицируемого этим моникером. Каждый моникер идентифицирует только один экземпляр объекта.

Типичный запрос, осуществляемый клиентом через моникер, будет выглядить следующим образом: «Создать и инициализировать объект, на который ссылается данный моникер.» Моникер создает и инициализирует объект, возвращает указатель на заданный клиентом интерфейс вновь запущенного объекта и затем (как правило) умирает в результате того, что клиент уменьшает его счетчик ссылок до 0 [24].

4.3.2. Распределенная объектная модель компонентов (DCOM)

COM с самого начала разрабатывалась как распределенная система, в которой компоненты можно создавать на другом хосте и их методы по сети, реально распределенная COM (Distributed COM — DCOM) появилась в 1996 г. DCOM можно рассматривать как незначительное расширение оригинальной COM. По существу DCOM добавляет к COM всего три основных элемента: способ создания удаленного объекта, протокол вызова методов удаленного компонента и механизмы обеспечения безопасного доступа к удаленному компоненту.

Технология, выпущенная в 1996 г., основана на использовании DCE/RPC. Она решала задачу вызова методов компонентов, размещенных на других хостах. В рамках DCOM имеется возможность установки безопасности посредством разрешения пользователям, работающим на определенных хостах, создавать удаленные объекты.

Опыт эксплуатации DCOM показал, что она обладает рядом серьезных недостатков, в частности, DCOM крайне сложна в настройке при наличии межсетевых экранов (Firewalls), имеются серьезные проблемы с безопасностью, сложности поддержки транзакций.

4.3.3. Технология COM+

Технология COM+ появилась в рамках Windows 2000. Она основывается, с одной стороны, на DCOM, а с другой стороны, на сервере транзакций (Microsoft Transaction Server). Основной целью создания COM+ следует считать разработку компонентной модели, которая могла эффективно использоваться в ИС крупного предприятия.

Отличительными особенностями среды COM+ являются следующие:

- наличие эффективных механизмов работы с транзакциями;
- возможность реализации асинхронного взаимодействия с помощью очередей сообщений;
- наличие механизмов работы с событиями;
- улучшенные показатели безопасности.
- возможность работы с пулом объектов.

Все это позволяет создавать достаточно хорошо масштабируемые, стабильно работающие приложения уровня крупного предприятия.

Объекты COM+, являющиеся экземплярами компонент COM+, работают под управлением среды COM+. Набор связанных между собой компонент COM+, находящихся в одной динамической библиотеке, называется приложением COM+. Приложение COM+

включает множество компонент и ролей для доступа к ним. Информация о всех зарегистрированных приложениях находится в каталоге COM+.

В 2002 г., т. е. всего через два года после появления COM+, была официально выпущена платформа Microsoft.NET, которая была объявлена Microsoft рекомендуемой основой для создания приложений и компонентов под Windows, в рамках которой используется собственная компонентная модель, радикально отличающаяся COM.

Очевидно, что фирма Microsoft не могла отказаться от анонсированной двумя годами ранее технологии и поэтому в. NET включены средства, позволяющие обращаться к компонентам COM из приложений .NET, и наоборот. По утверждению разработчиков COM+ и .NET являются отлично взаимодополняющими технологиями. Платформа .NET будет рассмотрена ниже.

Основные элементы технологии COM+. Технология COM+ представляет собой совокупность программных средств, которые обеспечивают функционирование и разработку распределенных приложений, преимущественно для сетей интранет.

Технология (платформа) COM+ включает в себя следующие основные элементы:

- программное обеспечение промежуточного уровня (middleware), обеспечивающее поддержку транзакций;
- интерфейсы прикладного программирования;
- утилиты, предназначенные для управления транзакциями.

Типовая программная модель приложений, использующих компоненты COM+, представляет собой трехзвенную архитектуру, которая включает в себя серверы, клиенты и промежуточное программное обеспечение. Бизнес-логика приложения сконцентрирована в объектах транзакций, а ПО промежуточного уровня, управляющее этими объектами, построено с использованием компонентной модели.

Компоненты COM+ строятся на базе COM-компонентов, т. е. должны поддерживать интерфейс IUnknown. Кроме того, компоненты COM+ должны поддерживать интерфейс IObjectcontrol и содержать ссылку на библиотеку типов. Компонент COM+ реализуется в составе внутреннего сервера, т. е. как dll.

Платформа COM+ в основе своей ориентирована на эффективную реализацию *транзакций*. Важной особенностью объекта COM+ является его способность существовать исключительно в рамках своей отдельной транзакции.

Атрибуты транзакций можно устанавливать тремя разными способами: жестко «зашить» в код на этапе разработке, с помощью редактора библиотеки типов или с помощью специальной утилиты, которая называется MTS Explorer. Каждая транзакция должна быть завершена в течение определенного промежутка времени, длительность которого задается в специальном параметре (по умолчанию —

это 1 минута). Если транзакция не завершается в течение указанного промежутка времени, то она отменяется.

Для управления транзакцией сервер транзакций автоматически создает объект, который называется объектом контекста транзакции или *контекстом объекта СОМ+*. Объект контекста поддерживает интерфейс IObjectcontext.

Одним из важных сервисов, предоставляемых СОМ+, является *сервис безопасности*. Данная платформа позволяет обеспечивать доступ к компонентам в зависимости от прав, которыми обладает клиент. Механизм обеспечения безопасности в СОМ+ включает декларативную и программную защиту данных.

Оба механизма основаны на использовании ролей. Роли можно рассматривать как абстрактное представление некоторой совокупности пользователей, в качестве которых могут выступать как отдельные группы пользователей, так и группы пользователей.

Для работы с ролями используется упоминавшаяся выше утилита MTS Explorer, с помощью которой администратор может создавать требуемые роли и ставить им в соответствие пользователей и группы. За каждой ролью закрепляются некоторое множество прав. Для аутентификации используются средства аутентификации Windows.

Декларативная защита данных заключается в ограничении доступа к тому или иному объекту пакету для пользователей и групп, которым поставлены в соответствие определенные роли. Декларативная защита настраивается средствами утилиты MTS Explorer. По умолчанию используются две роли: администратора (administrator) и читателя (reader).

Программная защита реализуется посредством использования объекта, который реализует интерфейс контекста (IObjectcontet). Данный интерфейс, в частности, реализует методы issecurityenabled и iscallerinrole. Программная защита реализуется на этапе разработки приложения. При попытке обращения клиента к некоторому объекту вызывается метод iscallerinrole, в качестве параметра вызова передается идентификатор роли. Если доступ к данному объекту для данной роли разрешен, то клиенту возвращается true.

При работе нескольких объектов СОМ+ в одном процессе, то метод iscallerinrole всегда возвращает true, поэтому если требуется обеспечить более точную идентификацию, то используется метод issecurityenabled.

Управление ресурсами. Для управления ресурсами используются три механизма:

- активизация «на лету» (just-in-time);
- пулинг объектов (object pooling).
- пулинг ресурсов (resource pooling);

Активизация «на лету» — это механизм, обеспечивающий возможность дезактивации и повторной активации. Повторную активацию можно выполнять, пока клиент сохраняет ссылку на соот-

ветствующий объект. Наличие данного механизма полезно в случае, когда объект используется эпизодически.

Пулинг объектов состоит в том, что при выполнении приложений в среде COM+ создается специальный пул объектов, для управления которым используется iobjectcontrol. Если COM+ объект предназначен для использования в пулинге, то метод canbepooled-интерфейса iobjectcontrol возвращать значение true. После деактивизации такого объекта COM+ сервер помещает его в пул. Объекты внутри пула доступны для немедленного использования любыми другими запросами клиентов. В случае если объект запрошен, но пул объектов пуст, то автоматически создает новый экземпляр объекта.

Пулинг ресурсов заключается в том, что ресурсы становятся доступными другим серверным объектам, после того как текущий COM+ объект дезактивируется. Прежде всего это касается соединений с базой данных, поскольку открытие и закрытие соединения с базой данных занимает много времени. Многократное повторение этой операции различными объектами COM+ применительно к одной базе данных может вызывать очень существенные временные задержки.

Пулинг ресурсов обеспечивает возможность нескольким объектам работать с ресурсом, при этом соединение устанавливается только один раз. Для реализации механизма пулинга ресурсов в COM+ используется распределитель ресурсов.

Оптимизация работы с COM+. Разработать приложение, использующее COM+, просто. Более сложной задачей является проектирование приложения таким образом, когда оно не только работает, но и приносит максимальный эффект от своего функционирования, воплощая все возможности используемой технологии. Для этого надо хорошо представлять механизм работы объектов COM и принципы работы с транзакциями.

Для исключения влияния транзакций друг на друга в COM+ реализуется система изоляции транзакций на высоком уровне. До тех пор пока данные, участвующие в одной транзакции не будут обработаны, они недоступны для другой транзакции. При этом, однако, снижается производительность системы.

Одним из наиболее важных понятий, относящихся к программированию для COM+, является понятие «активности» (activity), которое иногда называют действием. Активность можно определить как совокупность объектов, которые действуют совместно в интересах конкретного клиента. Активность может содержать объекты из разных пакетов. Каждый COM+-объект может принадлежать только одной активности, хотя каждая конкретная активность может содержать несколько объектов. Каждая транзакция может существовать только в одной активности, но активность может содержать более одной транзакции.

Программирование для COM+ предполагает, что объекты COM+ не должны разделяться между активностями [10].

4.3.4. .NET-компоненты

.NET Framework — программная платформа компании Microsoft, предназначенная как для создания обычных программ и веб-приложений. .NET является патентованной технологией.

В основу .NET Framework была положена амбициозная идея сделать платформно независимую универсальную виртуальную машину, которая могла бы выполнять код, написанный на произвольном языке программирования в различных ОС без перекомпиляции кода. Однако со временем Microsoft ограничилась поддержкой только собственных ОС, предоставив независимым разработчикам заниматься поддержкой других платформ.

Основными составными частями .NET Framework являются инвариантная к языку программирования среда исполнения (common language runtime, CLR) и библиотека классов Framework (framework class library, FCL). CLR — это некоторая обертка для API ОС, которая служит средой для исполнения управляемых приложений (managed applications). FCL предоставляет объектно-ориентированный API, к которому обращаются управляемые приложения.

При работе с управляемыми приложениями программист теряет возможность работать с Windows API, MFC, ATL, COM и другими знакомыми инструментами и технологией и должен работать с FCL. Если программист не хочет отказываться от «старых» технологий, то ему предоставляется возможность создавать приложения, основанные на использовании неуправляемого кода без использования CLR. Совмещение в рамках одного приложения управляемого и неуправляемого кодов возможно, но не приветствуется разработчиками платформы.

Инвариантная к языку программирования сфера достигается за счет того, что среда разработки создает байт-код, который интерпретируется виртуальной машиной. В качестве основных языков, поддерживаемых платформой NET, выступают C#, VB.NET, JScript.NET, C++/CLI, IronPython, IronRuby и F# (функциональный язык общего назначения).

В качестве входного языка виртуальной машины в .NET используется Common Intermediate Language (CIL). В более ранних версиях он назывался Microsoft Intermediate Language (MSIL).

Применение байт-кода позволяет получить кроссплатформенность на уровне скомпилированного проекта, который называют сборкой. Функцию преобразования сборки в исполняемый код целевого процессора реализует JIT-компилятор (just in time), который выполняет компиляцию «на лету».

Кроме того, имеется утилита, которая компилирует сборку в родной (native) код для выбранной платформы.

Данный подход идентичен используемой в Java виртуальной машине. Опыт использования виртуальных машин, как в рамках платфор-

мы Java, так и в рамках .NET показал, что их применение не приводит к резкому падению скорости работы приложений, а часто оказывается близким. Этот феномен можно объяснить тремя причинами. Во-первых, постоянно растет производительность аппаратных средств. Во-вторых, большая часть исполняемого кода — это библиотечный, т.е. уже скомпилированный код. В третьих, постоянно повышается скорость работы компиляторов.

Microsoft начала разрабатывать .NET Framework в конце 1990-х гг. и первая версия .NET 1.0 была выпущена в начале 2002 г.

На момент написания данной книги основной являлась версия 4.0, которая поддерживается средой разработки Microsoft Visual Studio 2010.

Практически каждая новая версия включала в себя новые технологии. Стек технологий .NET показан на рис. 4.8.

Назначение Среды выполнения и Базовой библиотеки классов описано выше.

Система построения клиентских приложений (Windows Presentation Foundation, WPF) предназначена для создания как автономных, так и запускаемых в браузере приложений.

WPF представляет собой векторную систему визуализации и широко использует язык XAML (Extensible Application Markup Language), представляющий собой XML, в котором реализованы классы .NET Framework. Кроме того, используются такие элементы, как элементы управления, привязка данных, макеты, двухмерная и трехмерная графика, анимация, стиль, шаблоны, документы, текст, мультимедиа и оформление.

Основой WPF является DirectX, что позволяет использовать аппаратные ускорители.

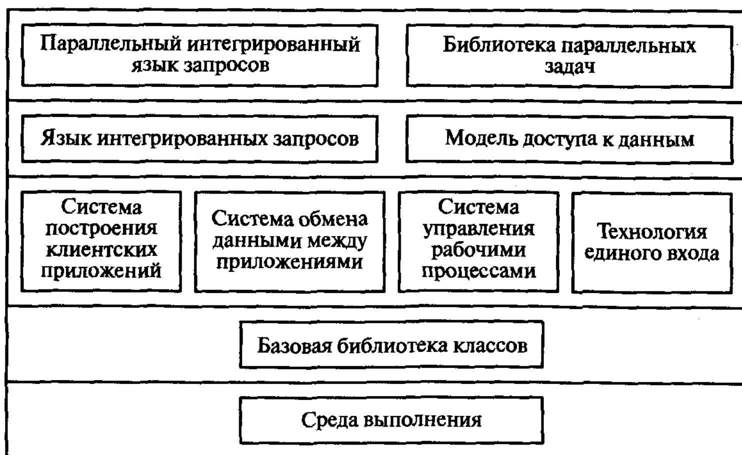


Рис. 4.8. Стек технологий .NET

Система обмена данными между приложениями (Windows Communication Foundation, WCF) представляет собой программный фреймворк, используемый для обмена данными между приложениями и входящей в состав .NET Framework.

WCF позволяет комбинировать функциональность разных технологий: таких как ASP.NET, Web Services, .NET Remoting, .NET Enterprise Services и System Messaging для создания распределенных приложений.

Система управления рабочими процессами (Windows Workflow Foundation, WF) — технология для определения, выполнения и управления рабочими процессами.

WF использует визуальное программирование, и в ее основе лежит декларативная модель программирования.

Использование WF предлагает три способа описания:

- последовательный процесс (Sequential Workflow);
- описание процесса в терминах конечный автомат;
- описание процесса с помощью правил.

Технология единого входа (Windows CardSpace, WCS) — это система идентификации пользователей при работе с разными ресурсами без необходимости повторного ввода имен и паролей. Данная технология является собственностью Microsoft, что затрудняет ее использование при работе со сторонними приложениями.

Windows CardSpace — патентованная технология единого входа от Microsoft. WCS — это способ простой и безопасной идентификации пользователей при перемещении между ресурсами Интернета без необходимости повторного ввода имен и паролей.

Модель доступа к данным (ADO.NET (ActiveX Data Objects.NET)) — интерфейс программирования приложений для доступа к данным, основанный на технологии компонентов ActiveX. и позволяющий представлять данные из разнообразных источников (реляционных баз данных, текстовых файлов и т.д.), в объектно-ориентированном виде.

Язык интегрированных запросов (*Language Integrated Query, LINQ*) позволяет поддерживать механизм запросов для коллекций объектов в памяти, реляционных баз данных и данных в формате XML; обладает расширяемой архитектурой, которая позволяет сторонним разработчикам реализовать доступ к их хранилищам данных через механизм LINQ.

Параллельные расширения (Microsoft Parallel Extensions for.Net) включает два компонента:

- библиотеку параллельных задач (Task Parallel Library, TPL);
- язык параллельных интегрированных запросов ((Parallel Language-Integrated Query, PLINQ).

Параллельные расширения ориентированы на работу с управляемым кодом и позволяют распараллеливать задачи, при работе в мультипроцессорных системах с общей памятью и, в частности, при использовании многоядерных процессоров.

Библиотека параллельных задач: в качестве базового понятия используется понятие задача (Task). Задача — это фрагмент кода, который может выполняться независимо от других фрагментов (задач). Как PLINQ, так и TPL, предлагают API для создания задач. При этом PLINQ разбивает запрос на отдельные запросы (задачи), а TPL разбивает задачу, циклы или последовательность блоков кода на подзадачи. Для синхронизации имеются соответствующие примитивы.

Параллельный интегрированный язык запросов (Parallel Language-Integrated Query, PLINQ) является параллельной реализацией LINQ. В отличие от LINQ PLINQ поддерживает параллельное выполнение запросов, используя все доступные ядра/процессоры.

Для поддержки параллельных расширений используется менеджер задач, на который возлагаются функции планирования. Планирование осуществляется на уровне потоков (нитей).

Компонент .NET представляет собой перекомпилированный MSIL, который может быть помещен в любой другой MSIL-компонент или клиентскую программу без выполнения компиляции.

Компонент .NET представляет собой перекомпилированный самоописываемый MSIL-модуль, построенный на базе одного или более классов или модулей, расположенных в DLL-файле сборки (DLL assembly file) [9].

Сборки представляют собой базовые строительные блоки, обеспечивающие развертывание и выполнение приложений .NET.

Исполняемый файл, библиотека или файл ресурсов, созданный на платформе .NET, представляет собой сборку. Кроме того, сборка обеспечивает реализацию механизма работы с версиями, область действия типов и ресурсов, зависимости и управление правами доступа.

Принципиальным отличием сборок от EXE- и DLL-файлов является то, что метаданные, описывающие содержимое, при работе со сборками хранятся внутри сборок, а при работе с EXE- и DLL-файлами метаданные хранятся в отдельных файлах.

Сборка представляет собой совокупность элементов, образующих сегмент приложения. Обычно в состав приложения входит несколько сборок. Сборка состоит из четырех элементов.

- манифеста;
- метаданных;
- кода приложения;
- ресурсов.

Манифест (manifest) содержит имя сборки, информацию о версии, информацию о файлах, входящих в состав сборки и безопасности.

Метаданные (metadata) — это информация о типах, объявленных в сборке, имена типов, их областях видимости, базовые классы и реализуемые интерфейсы.

Код приложения — это код, который компилируется в формат промежуточного языка Microsoft (Microsoft Intermediate Language, MSIL) и реализует типы, описываемые метаданными.

Ресурсы — это графические файлы, курсоры и статический текст.

Наличие в сборке метаданных и декларации означает, что сборка является полностью самодокументированной.

Кроме того, сборка также может содержать неуправляемый код и неуправляемые данные. Существует множество способов группировки элементов в сборке. Обязательным элементом сборки является манифест.

Сборка может представлять собой либо один физический файл, либо несколько файлов. Чаще всего сборка оформляется в виде одного файла. Такой файл называют переносимым исполняемым файлом (Portable Executable — PE). Сборки, состоящие из одного файла, обычно включают декларацию, метаданные и код MSIL.

Если сборка содержит несколько файлов, ее элементы размещаются в отдельных модулях. Модули могут состоять из откомпилированного кода MSIL, ресурсов. Модули, которые образуют многофайловую сборку, хранятся в файловой системе как отдельные файлы и связываются только логически через декларацию сборки, где содержится детальная информация о файлах, составляющих сборку.

Типовая структура однофайловой сборки показана на рис. 4.9, а многофайловой — на рис. 4.10.

Среда CLR работает с многофайловой сборкой как с единым объектом.

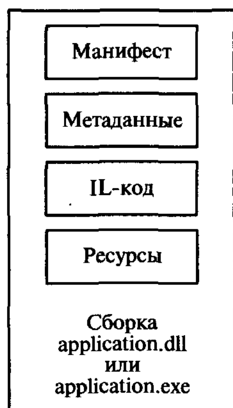


Рис. 4.9. Типовая структура однофайловой сборки

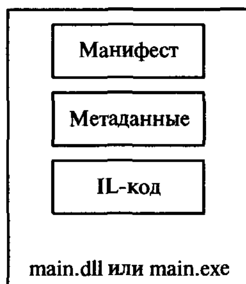


Рис. 4.10. Типовая структура многофайловой сборки

Лишь один из модулей сборки может содержать декларацию. При этом возможна ситуация, когда единственный модуль включает только декларацию. Но чаще всего декларация оказывается упакованной в один из связанных модулей.

В.NET имеется много разных типов компонентов. В частности, можно выделить визуальные и невизуальные компоненты. Визуальные компоненты имеют пиктограммы и в процессе разработки их можно перетаскивать в проект.

Невизуальные компоненты, или .NET component, могут работать на стороне клиента, сервера или в составе промежуточного ПО. Независимо от того, где он находится, .NET component всегда предоставляет своему клиенту некоторый сервис. В качестве клиента может выступать как другой компонент, так и клиентское приложение.

.NET component может быть либо локальным (local component), либо распределенным (distributed component). Локальный компонент (.dll) доступен локально (внутри домена приложения) на одном хосте. Удаленный (распределенный) компонент может быть доступен удаленно из других доменов приложений на том же или другом хосте. Домен приложений — это легковесный процесс, который может быть запущен и остановлен независимо от других легковесных процессов.

Следует заметить, что компонент не может напрямую обращаться к компоненту, находящемуся в другом домене приложений, поскольку каждому домену приложений соответствует собственное адресное пространство.

.NET DLL component может быть установлен как частный компонент (private component), который может работать только с определенным клиентом, либо как разделяемый (shared public component), которому неизвестен клиент. DLL-компоненты можно включать в любые приложения.

4.4. Технология CORBA

Основы CORBA. CORBA (Component Object Request Broker Architecture) — это стандарт, определенный группой компаний OMG (Object Management Group), который позволяет различным программным компонентам, написанным на разных языках программирования и работающим на разных машинах, взаимодействовать друг с другом.

Спецификация CORBA лежит в основе всех стандартов, разработанных OMG, и определяет механизмы взаимодействия приложений в распределенной системе.

Ключевыми компонентами данного стандарта являются объектный брокер запросов и язык определения интерфейсов IDL. Следующие по важности элементы — сервис динамического формирования

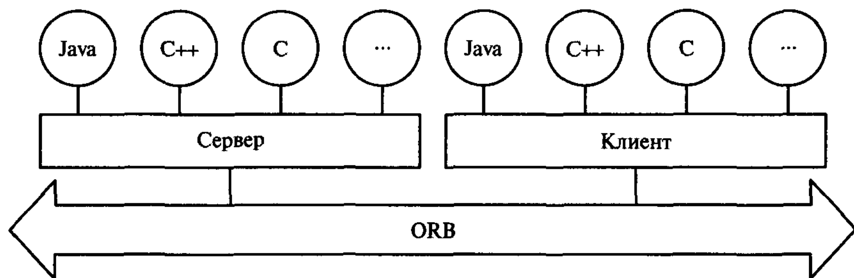


Рис. 4.11. Обобщенная структура системы, построенной с использованием CORBA

запросов, репозиторий интерфейсов, сервис динамической обработки запросов и сервис, обеспечивающий взаимодействие различных брокеров запросов.

Приложение представляется в виде множества объектов. Каждый из объектов, содержащих информацию о возможностях кода и способах его вызова, может быть вызван локально или через сеть из других приложений или объектов CORBA.

Обобщенная структура системы, построенной с использованием CORBA, показана на рис. 4.11. Основу CORBA-системы составляет системная шина, которая позволяет компонентам, реализованным на разных языках и работающим на разных платформах, находить друг друга и взаимодействовать. Эту шину называют брокером объектных запросов (Common Object Request Broker Architecture) или просто ORB.

Интерфейсы в CORBA определяются средствами языка IDL (Interface Definition Language — язык описания интерфейса). IDL описание используется для генерации кода заглушек для поддерживаемых языков программирования (Java, C++, C, COBOL, Lisp, PL/I, Smalltalk, Python). Сгенерированный код служит основой для доступа к объекту в определенной программной среде.

Данные вызова преобразуются в сетевой формат, сериализуются с помощью сгенерированных для объекта заглушек (Stubs) и передаются клиентским ORB (Object Request Broker — посредник вызова объектов) серверному ORB. На сервере данные вызова десериализуются с помощью сгенерированных для объекта скелетонов (*Skeletons*), где вызов выполняется. Данные, полученные при обработке вызова, передаются по цепочке обратно клиентскому коду.

История CORBA. Стандарт CORBA 1.0 был представлен в октябре 1991 г. группой компаний OMG. OMG — это международная открытая независимая организация, основанная в 1989 г. одиннадцатью компаниями (American Airlines, Canon Inc., Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems и др.) в целях создания общей среды для разработки объектно-ориентированных приложений

через разработку рекомендаций и детализированных спецификаций для объектно-ориентированного управления процессами.

Первые версии стандарта CORBA включали в себя базовые определения объектной модели CORBA, языка IDL, API для динамичного управления вызовами объектов (DII, Dynamic Interface Invocation) и репозитория интерфейсов (Interface Repository), а также концепцию базового адаптера объектов (BOA, Basic Object Adapter) — посредника между объектом и ORB. Единственным языком, официально поддерживаемым первыми версиями стандарта, стал язык программирования C. Одним из ограничений первых версий спецификации было отсутствие определения стандартного протокола взаимодействия различных ORB, что часто делало невозможным взаимодействие ORB от разных производителей. В 1996 г. появился стандарт CORBA 2.0, который устранил данный недостаток, представив протоколы GIOP (General Inter-ORB Protocal — общий протокол взаимодействия ORB) и IIOP (Internet Inter-ORB Protocol — протокол взаимодействия ORB через сеть Internet). Появилась возможность работы с языками C++ и Smalltalk и многое другое. В 1997—2001 гг. были добавлены поддержка языков Cobol, Ada и Java, концепция портативного адаптера объектов (POA, Portable Object Adapter) — еще один шаг на пути к стандартизации взаимодействия между различными реализациями CORBA. Кроме того, были включены поддержка взаимодействия с DCOM, служба сообщений (Messaging specification), минимальный стандарт CORBA (Minimum CORBA), система CORBA реального времени (Real-time CORBA) и ряд других служб.

Фактически, это последняя выпущенная версия. С современным состоянием развития технологии CORBA можно ознакомиться на официальном сайте www.corba.org.

Наибольшая популярность технологии CORBA пришлась на последние годы XX в. и первые годы XXI в. CORBA всегда рассматривалась как «тяжелая» технология, ориентированная на решение сложных задач, преимущественно в сфере крупных корпоративных систем. Однако CORBA далеко не всегда используется исключительно в крупных приложениях. Специализированные версии CORBA до сих пор работают в системах реального времени и малых встраиваемых системах.

CORBA, являясь спецификацией, предоставляет описание для реализации конкретных продуктов. Существует множество компаний и сообществ, поставляющих продукты CORBA для различных языков программирования. Список поставщиков можно найти на сайте CORBA.

Архитектура CORBA. Глобальная архитектура CORBA восходит к модели OMG, в рамках которой выделяется четыре группы архитектурных элементов (рис. 4.12):

- брокер объектных запросов — ORB;
- сервисы CORBA;

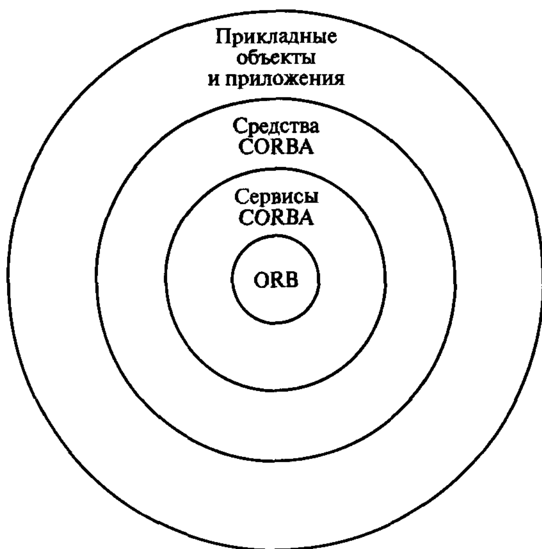


Рис. 4.12. Группы архитектурных элементов CORBA

- средства CORBA;
- прикладные объекты и приложения.

Брокер объектных запросов (Object Request Broker — ORB) определяет объектную шину CORBA. Сервисы CORBA (CORBA Services) определяют системный уровень объектной структуры и могут рассматриваться как расширение объектной шины. Средства CORBA (CORBA Facilities) определяют сервисы, которые непосредственно используются приложениями (бизнес-объектами). Application Objects представляют прикладные объекты и приложения, т. е. конечных потребителей инфраструктуры CORBA.

Каждый ниже лежащий уровень выступает как сервер для выше лежащего.

В основе CORBA лежит понятие «объектная модель». ***Объектная модель*** описывает объектные концепции и терминологию, применимые к клиентскому коду и реализации объектов. Объектная модель CORBA — пример классической объектной модели, в которой клиенты посылают сообщения объектам, а объекты интерпретируют сообщения и выполняют соответствующие им действия.

Объект (object) — идентифицируемая, инкапсулированная сущность, предоставляющая один или более сервисов, которые могут быть запрошены клиентами.

Клиенты запрашивают службы объектов через специальные *запросы* (request). Информация, содержащаяся в запросе, включает в себя объект, название операции, параметры и необязательный контекст запроса. Параметры могут быть входящими (input), выходя-

щими (output) и двунаправленными (input-output), а необязательный контекст представляет собой маппинг из строк в строки. Результатом запроса является выполнение сервиса для клиента, а также возвращение результирующего значения (если таковое определено для операции) и значений в output и input-output параметрах. При ошибочном выполнении вызова клиенту возвращается *исключение* (exception).

Объекты создаются и уничтожаются. Создание и уничтожение объектов происходит вследствие выполнения запросов. Клиент получает доступ к созданным объектам через *ссылки* (reference). CORBA разграничивает понятие реализации объекта и ссылки на объект. Реализация объекта представляет собой серверный код объекта с определенными операциями, в то время как ссылка идентифицирует объект для клиента и используется для осуществления вызовов. Множество различных ссылок может обозначать один объект.

Типы (type) в CORBA определяются предикатной логикой. Сущность, которая удовлетворяет предикату, определенному для типа, считается *членом* (member) данного типа. *Объектные типы* (object type) — типы, чьими членами являются объектные ссылки. Объектные типы образуют единую иерархию с типом Object в качестве корня, при этом любой тип может использоваться везде, где используются его предки. Объектные типы могут быть параметрами и результирующими типами операций, а также компонентами других структурированных типов. *Необъектные типы* также определены в CORBA и описываются через конструкции IDL. В качестве базовых типов (basic types) определены булевой тип, различные числовые и строковые типы, а также тип any, обозначающий любой базовый тип. Базовые типы, как и объектные, могут использоваться в качестве компонентов для множества различных структурированных типов, включая структуры, массивы, последовательности и объединения.

Одним из ключевых понятий в объектной модели CORBA является *интерфейс* (interface). Интерфейс представляет собой описание операций, предоставляемых объектом для вызова клиенту, а также может содержать определение типов, используемых для данных операций. Интерфейсы в CORBA образуют иерархию и соответствуют иерархии объектных типов. Объекты в CORBA могут реализовывать более одного интерфейса.

Язык описания интерфейсов (IDL) определяет типы объектов посредством спецификации их интерфейсов. Интерфейс состоит из множества именованных операций и их параметров. Описание на IDL обеспечивает ORB всей необходимой информацией о типе объекта.

Язык описания интерфейсов рассматривается как средство, с помощью которого реализация объекта сообщает своим потенциальным клиентам о том, какие операции доступны и каким образом их следует вызывать. Описание на языке IDL транслируется в код на конкретном языке программирования.

Брокер объектных запросов. Брокер объектных запросов представляет собой объектную шину, которая позволяет объектам прозрачно генерировать запросы и получать ответы от локальных или удаленных объектов. Причем клиент ничего не знает о механизмах для создания и сохранения состояния. Поскольку CORBA — это только спецификация, которая может быть реализована разными поставщиками, то начиная с CORBA 2.0 определяется взаимодействие (interoperability) ORB разных производителей.

CORBA ORB позволяет объектам обнаруживать друг друга во время выполнения («на лету») и вызывать сервисы друг друга, т.е. реализует типовой спектр сервисов распределенного промежуточного программного обеспечения, однако ORB намного более сложен, чем, например, вызовы удаленных процедур, в частности:

- поддерживает как статические, так и динамические вызовы методов;
- отделяет интерфейс от его реализации;
- является самоописываемой системой;
- создает прозрачность относительно местоположения объектов.

Все ORB позволяют реализовывать как статические, так и динамические вызовы методов.

Статические вызовы методов определяются CORBA ORB во время компиляции, а динамические вызовы — во время выполнения.

К достоинствам статического вызова методов можно отнести наличие строгого контроля типов на стадии компиляции и высокую скорость работы, а динамического — максимальную гибкость при отложенном (на этапе выполнения) связывании. (Следует отметить, что большинство других видов middleware поддерживают только статическое связывание.)

Отделение интерфейса от его реализации предоставляет независимые от языка типы данных, что дает возможность вызывать объекты из любого языка и для любой операционной системы.

CORBA как *самоописываемая система* предоставляет метаданные на этапе выполнения для описания каждого серверного интерфейса, известного системе. Каждый CORBA ORB должен поддерживать репозитарий интерфейсов (Interface Repository), который содержит текущую информацию, описывающую предоставляемые сервером функции и их параметры. Клиенты используют метаданные, чтобы определить, каким образом вызывать сервисы во время выполнения. Метаданные также помогают инструментальным средствам создавать код «на лету». Такие метаданные генерируются автоматически либо прекомпиляторами языка IDL, либо такими компиляторами, которые знают, как генерировать IDL непосредственно из объектно-ориентированного языка.

Прозрачность относительно местоположения объектов поддерживается взаимодействием между объектами, расположенными в одном процессе, на одном хосте и на разных хостах одной локаль-

ной сети. Если требуется организовать взаимодействия с другими объектами, находящимися, например в Internet, то используются сервисы протокола IIOP (Internet Inter-ORB Protocol — Internet-протокола взаимодействия ORB), причем это делается прозрачно для пользователя.

ORB может выступать посредником при взаимодействии между «мелкозернистыми» объектами, по уровню сложности соответствующих например классам C++, так и для «крупно-зернистых» объектов, в качестве которых могут выступать, например бизнес-объекты.

Структура ORB показана на рис. 4.13.

ORB позволяет устанавливать клиент-серверные отношения между объектами в распределенной компьютерной среде, при этом один и тот же объект может выступать как в качестве клиента, так и в качестве сервера.

Для обращения к серверу клиент может использовать либо динамический, либо статический вызов.

При реализации динамических вызовов, т.е. вызовов, не использующих предварительно скомпилированные заглушки, клиент использует *интерфейс динамического вызова ИДВ* (Dynamic Invocation Interface), позволяющий обнаруживать методы, которые необходимо вызвать, во время выполнения. CORBA предлагает стандартный API для поиска метаданных, определяющих серверный интерфейс, генерации параметров, удаленного вызова и получения результатов. Указанные метаданные хранятся в *репозитарии интерфейсов РИ* (Interface Repository) — распределенной базе данных, которая содержит в машинном формате версии определенных на IDL интерфейсов.

При реализации статического вызова используются *стабы* (stub). Стабы клиента и сервера генерируются компилятором IDL. Клиент должен иметь стаб для каждого интерфейса, который содержит, в частности, код маршалинга. Для обращения к удаленному объекту клиенту достаточно просто вызвать метод. С точки зрения

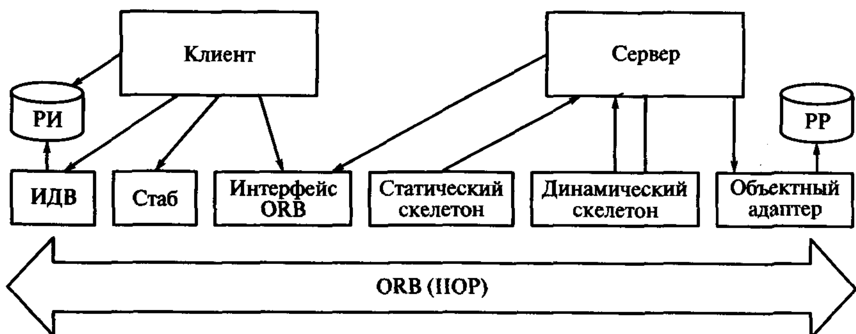


Рис. 4.13. Структура ORB

клиента стабы — это локальный заместитель (proxy) для удаленного объекта.

API Репозитария Интерфейсов (Interface Repository API) позволяет получать доступ описаниям интерфейсов зарегистрированных компонентов, поддерживаемых ими методов и их параметров. В CORBA такие описания называют сигнатурой метода (method signature).

Интерфейс ORB предлагает ряд полезных сервисов, в частности, API для преобразования ссылки на объект в строку и обратно, что может быть полезно в случае, если необходимо сохранять или пересылать ссылки.

ORB предоставляют глобальные идентификаторы — для уникальной и глобальной идентификации компонентов и их интерфейсов среди ORB разных производителей и множества репозитариев, которые генерируются системой и представляют собой уникальную строку для поддержания целостности в соглашениях по наименованию.

Сравнивая статические и динамические вызовы, следует отметить, что статические вызовы намного проще в программировании, они быстрее работают и самодокументируемы. Динамические вызовы предоставляют максимальную гибкость, но сложнее в программировании. Однако они незаменимы в том случае, когда требуется анализ выполнения.

Серверная часть не может определить разницу между статическим и динамическим вызовами, поскольку оба они имеют одинаковую семантику сообщения. В обоих случаях ORB находит объектный адаптер сервера, передает ему параметры, а затем передает управление коду объекта с помощью стаба, который в CORBA называется скелетоном (skeleton) сервера.

На стороне сервера размещаются статические скелетоны, динамические скелетоны и адаптеры объектов.

Статические скелетоны предоставляют статические интерфейсы каждому из реализуемых сервисов. Эти стабы создаются компилятором IDL и определены для каждого класса объектов.

Динамические скелетоны предоставляет механизм связывания реального времени для тех серверов, которым необходимо управлять входящими вызовами методов компонентов, не имеющих скомпилированных скелетонов. Динамический скелетон находит значения параметров вызова во входном сообщении и определяет, для какого объекта и метода они предназначены.

Динамические скелетоны используются, в частности, для реализации универсальных мостов между ORB, интерпретаторами и языками сценариев, чтобы динамически сгенерировать реализацию объекта. Динамический скелетон может обрабатывать как статические, так и динамические вызовы клиентов.

Объектный адаптер (*Object Adapter*) принимает запросы к сервису со стороны серверных объектов и обеспечивает среду реального времени для создания экземпляров серверных объектов, передачи

запросов объектам и назначения объектам идентификаторов. Объектный адаптер также регистрирует поддерживаемые им классы и их экземпляры этапа выполнения (т.е. объекты) в *репозитарии реализаций РР* (Implementation Repository), который представляет собой репозитарий времени выполнения и содержит информацию о классах и объектах, экземпляры которых созданы.

CORBA предлагает два типа адаптеров объектов: базовый адаптер объектов (Basic Object Adapter, BOA) и переносимый адаптер объектов (Portable Object Adapter, POA).

Сервисы CORBA. В качестве базовых сервисов в рамках CORBA используется пятнадцать сервисов:

1) сервис именования (Naming);
2) объектный сервис жизненного цикла (Life Cycle);
3) сервис событий (Event Service);
4) сервис долговременного хранения (Persistence Service);
5) сервис контроля совместного доступа (Concurrency Control Service);
6) сервис транзакций (Transaction Service);
7) сервис отношений (Relationship Service);
8) сервис внешнего представления (Externalization Service);
9) сервис запросов (Query Service);
10) сервис лицензирования (Licensing Service);
11) сервис свойств (Properties Service);
12) сервис времени (Time Service);
13) сервис безопасности (Security Service);
14) сервис коммерции (Trader Service);
15) сервис контейнеров (Collection Service).

Объектные сервисы являются базовыми строительными блоками для создания распределенной объектной инфраструктуры. Они представляют собой следующий этап в последовательности шагов, направленных на достижение главной цели — создания распределенных компонентов, или, как их называет OMG, бизнес-объекта (business objects).

Базовые сервисы CORBA являются инфраструктурными сервисами, которые не связаны ни с одним предметным доменом.

Сервис именования — это механизм, используемый объектами ORB для обнаружения других ORB объектов. CORBA имена представляют собой идентификационные значения, которые определяют объект. Сервис именования отображает CORBA имена в объектные ссылки. Контекст имен (naming context) представляет собой пространство имен (namespace), в котором имена объектов уникальны. Каждый объект имеет уникальный ссылочный идентификатор (reference ID). Сервис именования был разработан на основе существующих служб имен и каталогов.

Объектный сервис жизненного цикла (Life Cycle) обеспечивает реализацию таких функций, как создание, копирование, переме-

щение и удаление объектов, может выполнять данные действия над группой связанных объектов.

Сервис событий (Event Service) позволяет компонентам, находящимся на шине, реагировать на события в системе. Объекты имеют возможность регистрировать интерес к определенным системным событиям или отменять такую регистрацию.

Сервис долговременного хранения (Persistence Service) — это служба, позволяющая создавать хранилища компонентов в различных средах, включающих плоские файлы и реляционные базы данных.

Сервис контроля совместного доступа (Concurrency Control Service) предоставляет собой механизм, который позволяет выполнять блокировки от имени транзакций и потоков выполнения.

Сервис Транзакций (Transaction Service) позволяет организовывать двухфазное завершение (two-phase commit) транзакций для компонентов, при этом имеется возможность работать не только с простыми (flat), но с вложенными (nested) транзакциями.

Сервис отношений (Relationship Service) позволяет создавать динамические ассоциации между компонентами, непосредственно не связанными друг с другом. Данный сервис используется преимущественно для обеспечения целостности.

Сервис внешнего представления (Externalization Service) обеспечивает стандартный способ для передачи данных компоненту и обратно, а также для пересылки самих компонентов по сети. По сути, это сервис сериализации.

Сервис запросов (Query Service) позволяет выполнять запросы для объектов, в частности находить объекты по их свойствам. В основе Сервиса запросов лежит SQL3 и язык объектных запросов (Object Query Language).

Сервис лицензирования (Licensing Service) позволяет отслеживать использование компонентов с целью получения платы.

Сервис свойств (Properties Service) предоставляет доступ к свойствам компонентов, позволяет, в частности, читать значения свойств, изменять значения свойств, изменять состав свойств.

Сервис Времени (Time Service) предоставляет интерфейсы для синхронизации часов в распределенной среде и опеспечивает возможность управлять событиями, привязанными к астрономическому времени.

Сервис безопасности (Security Service) позволяет поддерживать инфраструктуру для обеспечения безопасности системы распределенных объектов, включая такие механизмы, как аутентификацию, списки контроля доступа и конфиденциальность, и, кроме того, отвечает за делегирование прав доступа к объектами.

Сервис коммерции или трейдер-сервис (Trader Service) — это сервис «Желтых страниц» для объектов; позволяющий опубликовывать предлагаемые сервисы и находить их.

Сервис контейнеров (Collection Service) позволяет создавать контейнеры, в которые можно помещать компоненты и формировать из них контейнеры.

Средства CORBA. Средства делятся на горизонтальные и вертикальные.

Горизонтальные средства определяют интерфейсы, не зависящие от предметной области.

Горизонтальные средства включают следующие:

• средства пользовательского интерфейса (USER Interface) — инструменты для разработки интерфейса, средства для автоматизации разработки интерфейса, спецификации на рабочее пространство пользователя и т.д.;

• средства управления информацией (Information management) предоставляют операции, с помощью которых можно моделировать, описывать, сохранять, выбирать, перемещать информацию и обмениваться ею;

• средства управления системой включают множество утилит, реализующих функции системного администрирования, т.е. определяют интерфейсы основных операций, обеспечивающих управление, мониторинг, безопасность, конфигурирование и т.д.;

• средства управления задачами включают средства управления потоками работ (workflow facility), средства программных агентов (agent facility), средства управления правилами (rule management facility), средства автоматизации (automation facility).

Вертикальные средства предназначены для поддержания конкретных областей рынка: финансовой деятельности, промышленного производства, медицины и т.д. [11].

4.5. Технология Enterprise Java Beans

Технология Enterprise Java Beans (EJB) представляет собой высокоуровневый подход к построению распределенных приложений масштаба предприятия. EJB — это модель серверных компонентов (server component model) для Java.

Основная идея технологии Enterprise JavaBeans состоит в создании такой инфраструктуры для компонентов, чтобы они могли легко добавляться (plug in) и удаляться из серверов без перекомпиляции кодов приложения, тем самым расширяя или ограничивая функциональность сервера.

История Enterpise Beans. Фирма Sun Microsystems Inc. анонсировала технологию EJB в 1996 г., а в 1998 г. была представлена первая реализация (версия 1.0). На момент написания книги последней была версия EJB 3.0, которая появилась в 2006 г.

Версия. EJB 1.0 — это начальная версия, в рамках которой поддерживались stateful и stateless компонент, с которыми можно было

работать через удаленный интерфейс. В качестве желательной опции рассматривались компоненты, имеющие состояния.

В *версии EJB 1.1* в качестве обязательных появились компоненты с состоянием и XML-дескриптор (deployment descriptor).

В *версии EJB 2.0* в дополнение к удаленному интерфейсу появились локальный интерфейс, который могли использовать компоненты, работающие в том же контейнере; компоненты, управляемые сообщениями the message-driven bean (MDB), которые могли принимать асинхронные сообщения. Для компонентов с сохранением была определена возможность сохранения состояния средствами контейнера. Появился язык Enterprise JavaBeans Query Language (EJB QL), похожий по структуре на SQL и предназначенный для поиска компонентов.

В *версии EJB 2.1* была добавлена поддержка работы с Web-сервисами, расширились возможности EJB QL, и в дескрипторе развертывания вместо DTD стала использоваться XML schema.

Можно заметить, что основные различия между версиями состоят в наращивании возможностей.

Версия EJB 3 радикально отличается от предшествующих и представляет собой фактически новую компонентную модель. Основное отличие состоит в том, что пользователь не должен ничего знать о домашнем интерфейсе (Home Interface). В рамках данной спецификации присутствует идея POJO (Plain Old Java Objects), состоящая в том, что программист пишет только ту часть кода, которая реализует бизнес-логику. Для того чтобы определить, каким именно способом реализуется бизнес-код, используется механизм аннотаций. Например, если требуется определить интерфейс как локальный, достаточно описать интерфейс обычным способом и вставить строку аннотацию @Local. Механизм аннотаций позволяет существенно упростить процесс разработки.

Таким образом, речь идет о двух разных компонентных моделях и разных философиях разработки компонентов.

Использование EJB позволяет упростить процесс разработки КИС за счет того, что многие функции реализуются контейнерами. Кроме того, EJB можно использовать многократно, причем для повторного применения не требуется перекомпиляция всего проекта.

EJB-компоненты хорошо сопрягаются с Web-сервисами и с CORBA.

Архитектура Enterprise Java Beans.

Архитектурная модель EJB. Она включает в себя следующие элементы (рис. 4.14):

- JEE-контейнер;
- EJB-контейнер;
- Web-контейнер;
- клиента;
- базу данных (БД).

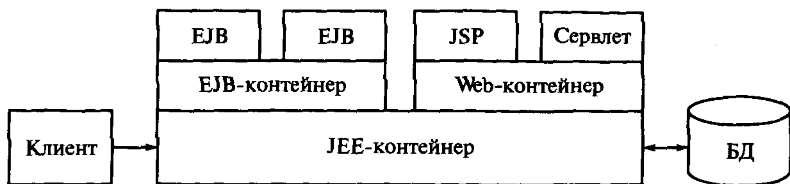


Рис. 4.14. Архитектурная модель EJB

JEE-контейнер представляет собой EJB-совместимый сервер приложений, внутри которого имеются EJB-контейнер и Web-контейнер. EJB-компоненты функционируют внутри EJB-контейнера, а сервлеты и JSP — внутри Web-контейнера. БД используется для хранения состояния EJB-компонентов. Для взаимодействия с контейнером EJB-компонент используют Home-интерфейс, а для взаимодействия с клиентом — Remote-интерфейс. Кроме того, в процессе функционирования EJB-компоненты пользуются такими сервисами, как JNDI, JTS, системы безопасности и т.п.

Рассмотрим основные компоненты архитектуры EJB.

JEE совместимый сервер приложений обеспечивает следующие основные сервисы: HTTP-сервис, сервисы безопасности, Java Naming and Directory Interface (JND) — сервисы.

EJB-контейнер реализует следующие сервисы: управление жизненным циклом компонента; управление транзакциями; управление персистностью.

EJB-контейнер поддерживает интерфейс для EJB-компонентов. Клиенты никогда не обращаются напрямую к EJB-компонентам. Все обращения идут к контейнеру, который выполняет функции посредника. На рис. 4.15 показана структура EJB-контейнера. Когда компоненты хотят обратиться к внешним компонентам, то используется информация, содержащаяся в контексте. EJB-контейнер реализует такие функции, как создание и уничтожение EJB-компонентов.

EJB-компоненты образуют распределенную компонентную систему. Приложения и другие компоненты обычно могут обращаться к EJB-компоненту через удаленный интерфейс. Если вызывающий и вызываемый компоненты находятся в одном контейнере, то ис-

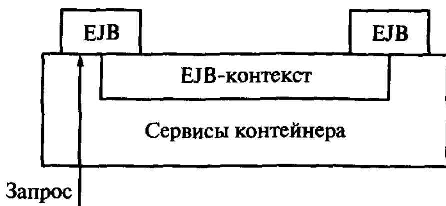


Рис. 4.15. Структура EJB-контейнера

пользуется локальный интерфейс. Для того чтобы поместить EJB-компонент в контейнер, необходимо отредактировать дескриптор размещения (Deployment Descriptor, DD), представляющий собой XML-документ.

Компонентная модель EJB 2.X. Каждый EJB-компонент имеет объектный интерфейс (EJB-Object), через который клиент может обратиться к данному компоненту. При этом клиенту могут быть неизвестны подробности, касающиеся, в частности, местонахождения компонента. Этот интерфейс называют удаленным (remote interface). Конкретный экземпляр EJB-компонента управляется контейнером через домашний интерфейс (home interface). Каждый EJB-компонент должен поддерживать как удаленный, так и домашний интерфейс.

Конфигурирование EJB-компонента осуществляется посредством редактирования конфигурационного файла. Структура EJB-компонента и процесс взаимодействия клиента и EJB-компонента показан на рис. 4.16.

EJB-класс реализует эти два интерфейса. Взаимодействие осуществляется следующим образом. Клиент обращается к компоненту по имени (Lookup), которое используется для получения объектной ссылки (Object Reference, OR) через JNDI-сервис. При обращении к контейнеру (Create) он создает экземпляр EJB-компонента и передает ему параметры вызова. Очевидно, что для того чтобы можно было получать ссылки, компоненты должны быть предварительно зарегистрированы с помощью JNDI-сервиса. Затем вызывается требуемый метод (Invoke).

EJB-компонент представляет собой программный компонент, который может быть повторно использован без перекомпиляции в разных приложениях. Для использования EJB-компонента достаточно

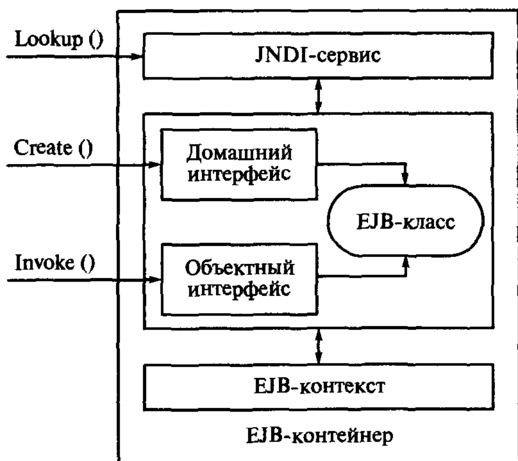


Рис. 4.16. Структура EJB-компонента

поместить его в соответствующий каталог и выполнить настройки конфигурационного файла, который называют Deployment Descriptor (DD). DD представляет собой XML-файл.

Компонентная модель EJB определяет три основных типа компонентов:

- сеансовые компоненты (Session Beans);
- компоненты-сущности (Entity Beans);
- компоненты, ориентированные на сообщения (Message Driven Beans).

В свою очередь, сеансовые компоненты делятся на две группы: без состояния (Stateless Session Beans) и с состоянием (Stateful Session Beans).

Сеансовые компоненты без состояния предназначены для выполнения бизнес-операций, когда не требуется сохранять внутреннее состояние компонента. К таким операциям можно отнести операции, связанные с поиском слов в словаре, архивирование файлов, калькуляторы. Компоненты данного типа можно реализовать в виде Web-сервисов.

Сеансовые компоненты с состоянием помнят свое состояние только в пределах сеанса. Время жизни таких компонентов соответствует одному сеансу, продолжительность которого может составлять от нескольких секунд до нескольких часов и даже дней.

Типичный пример использования сеансовых компонентов с состоянием — интернет-магазин. EJB-компонент может описывать, например содержимое корзины покупателя. Тогда сеансом будет считаться время с момента регистрации пользователя на сайте магазина до момента закрытия пользователем браузера или перехода к другому сайту.

Компоненты-сущности, напротив, способны поддерживать свое состояние. Их состояние сохраняется в реляционной базе данных. Каждому компоненту обычно ставится в соответствие строка базы данных. В рамках компонентной модели EJB 2.X выделяется два варианта управления состоянием компонента: ручное сохранение (Bean Managed Persistence, BMP) и автоматическое сохранение (Container Managed Persistence, CMP).

В первом случае ответственность за сохранение состояния возлагается на компонент, т.е. на программиста. Во втором случае сохранение и восстановление состояния возлагается на контейнер. Информацию, необходимую для работы с БД, контейнер берет из дескриптора размещения (Deployment Descriptor, DD), который представляет собой XML-файл.

При работе с компонентами-сущностями можно использовать EJB QL, который позволяет находить компоненты по первичному ключу или значениям отдельных полей.

Компоненты, ориентированные на сообщения, предоставляют асинхронный интерфейс. Такому компоненту можно отослать

сообщение и продолжать работу сразу после окончания его передачи, не дожидаясь окончания его обработки, призваны взаимодействовать с системами очередей сообщений, в первую очередь с JMS. Обычно компоненты, ориентированные на сообщения, выполняют роль адаптеров, принимающих асинхронные сообщения, что полезно при построении систем, ориентированных на работу с событиями.

Как для всякой компонентной системы, для EJB характерно выделение типовых ролей при разработке и сопровождении приложений, использующих EJB.

Применительно к EJB выделяются следующие роли:

- провайдер сервера EJB;
- провайдер контейнера EJB;
- разработчик EJB;
- специалист по внедрению/установке EJB;
- разработчик приложения.

Провайдер сервера EJB — это производитель EJB-совместимого сервера приложений. Производитель сервера EJB может также выступать в качестве производителя контейнера EJB.

Провайдер контейнера EJB — это производитель EJB-контейнера. Производитель сервера EJB обычно выступает в качестве производителя контейнера EJB. Как правило, это крупные фирмы или организации, такие как IBM или Oracle.

Разработчик EJB — это разработчик или группа разработчиков, которые имеют не только знания о спецификации EJB, но и бизнес-требования, так как он отвечает за кодирование бизнес-логики в серверных компонентах. Разработчик EJB может создавать EJB-компоненты как для нужд конкретной организации, так и на продажу. Спецификация EJB называет разработчика EJB провайдером EJB.

Специалист по внедрению/установке EJB ответственен за определение EJB, всех поддерживающих классов и их правильную инсталляцию на сервере EJB.

Разработчик приложения пишет клиентские приложения, используя EJB-компонентов. Обычно разработчик приложения имеет возможность подключать готовые EJB-компоненты без необходимости их разработки или тестирования.

Спецификация EJB называет разработчика приложения ассемблером приложения.

После того как разработка кода клиента и сервера завершена, выполняется их компиляция в файлы классов. EJB-компонент упаковывается в .jar-файл, который, в свою очередь, вместе с другими Web-компонентами (.war), DD-, XML-файлы и коды клиентской части (.jar) упаковывается в .ear-файлы.

DD-файл — это файл размещения в XML-формате. Этот файл создается обычно автоматически.

Ниже приведен фрагмент DD-файла:

```
<?xml version="1.0">
<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>accountBean</ejb-name>
            <home>accounthome</home>
        <remote>account</remote>
            <ejb-class>Account</ejb-class>
            <persistence-type>Container
              </persistence-type>
            <pri-key-class>Integer</pri-key-class>
            ...
            <cmp-field><field-name>id</field-name>
              </cmp-field>
            ...
            <cmp-field><field-name>name</field-name>
              </cmp-field>
    </enterprise-beans>
    <assembly-descriptor>
        <security-role>
        ...
        </security-role>
        ...
    </assembly-descriptor>
</ejb-jar>
```

Особенности компонентной модели EJB 3.0. Одним из побудительных мотивов создания EJB-технологии были сложности CORBA. Однако эволюция технологии EJB привела к тому, что эта технология стала казаться разработчикам чрезмерно сложной, в частности:

- тяжеловесная модель программирования, требующая работы с несколькими интерфейсами;
- необходимость непосредственно взаимодействовать с Java Naming Directory Interface (JNDI);
- необходимость работать с непомерно сложным XML DD.

Эти недостатки устранены в EJB 3.0, где используется только один бизнес-интерфейс вместо интерфейсов home и remote, минимизируется использование DD за счет использования аннотаций. Кроме того, используется новый механизм для сохранения состояния для компонентов-сущностей (Entity Beans). Одна из новых черт EJB 3.0 — использование Java Persistence API (JPA) для реализации сохранения состояния.

Сохранения состояния (персистенс) — это способность автоматически сохранять состояние Java объектов в реляционных базах данных.

JPA — это самостоятельная технология, которая обеспечивает объектно-реляционное отображение JAVA-объектов и предоставляющая API для сохранения, получения и управления такими объектами. Собственно JPA — это спецификация, которая представляет собой часть спецификации EJB 3.0. Наиболее известными реализациями JPA являются Hibernate, Oracle TopLink, Apache OpenJPA.

JPA состоит из трех элементов:

• API — интерфейсы в пакете javax.persistance, представляющем собой набор интерфейсов;

• JPQL — объектный язык запросов, похожий на SQL, но запросы выполняются к объектам;

• Metadata — аннотации над объектами, представляющими собой набор аннотаций, которыми описывают объектно-реляционные отображения. Метаданные можно описывать либо с помощью XML файлов, либо с помощью аннотаций.

Аннотирование — одна из ключевых особенностей программирования в технологии EJB 3.0. Эта возможность позволяет разработчикам аннотировать программные элементы непосредственно в исходных текстах на языке Java с целью управлять поведением или развертыванием компонентов.

Механизм аннотаций введен в язык Java, начиная с версии 1.5. Аннотации не меняют смысла программы, однако позволяют непосредственно в тексте программы помещать указания для различных инструментальных средств. Механизм аннотаций дает возможность вводить новые типы аннотаций и использовать существующие для аннотирования программных конструкций.

Ниже приведен пример применения механизма аннотаций. Интерфейс описывается как обычный Java интерфейс (Plain Old Java Interface, POJI). Класс компонента представляет собой обычный Java объект (Plain Old Java Object, POJO). Строка @Stateless представляет собой аннотацию, которая указывает на то, данный объект должен быть реализован как EJB-компонент без сохранения состояния. Для того чтобы, исполнить этот компонент, его надо поместить в EJB-контейнер.

```
package ejb3component.example;
public interface Hello {
     public void Hello(String name);
}
package ejb3component.example;
import javax.ejb.Stateless;
@Stateless
public class HelloBean implements Hello {
     public void Hello(String name) {
          System.out.println("Hello " + name);
     }
}
```

На данном примере достаточно хорошо просматривается идея POJO, которая состоит в том, что пользователь пишет только код, который реализует требуемую функциональность, а способ реализации указывает в аннотациях [20, 50].

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Охарактеризуйте понятие компонент.
2. В чем различие понятия «программный компонент» и «объект»?
3. Охарактеризуйте основные фазы развития технологий разработки распределенных систем.
4. Перечислите и охарактеризуйте известные вам компонентные технологии.
5. Каким образом реализуется вызов удаленной процедуры?
6. Что такое маршаллинг и демаршаллинг?
7. В чем состоят основные недостатки вызова удаленных процедур?
8. Что такое DCE?
7. Что такое служба каталогов?
8. Что такое служба распределенного времени?
9. Что такое и каким образом реализуется RMI?
10. Что такое COM?
11. Каково назначение интерфейса IUnknown?
12. Опишите процесс создания объектов COM.
13. Охарактеризуйте механизмы повторного применения объектов COM.
14. Каким образом применительно к COM реализуется управление перманентностью?
15. Что такое DCOM и COM+?
16. В чем различие COM, DCOM и COM+?
17. Что такое .NET Framework?
18. Опишите общие принципы функционирования .NET Framework.
19. Охарактеризуйте компонентную модель .NET.
20. Что такое CORBA?
21. Что такое брокер объектных запросов CORBA?
22. Охарактеризуйте объектную модель CORBA.
23. Перечислите и охарактеризуйте базовые сервисы CORBA.
24. Что такое EJB?
25. В чем состоит различие между EJB 1.x, EJB 2.x и EJB 3.x?
26. Охарактеризуйте понятия «сеансовые компоненты», «компоненты-сущности» и «компоненты, ориентированные на сообщения».
27. Опишите архитектурную модель EJB.
28. Поясните, каким образом EJB-компонент взаимодействует с EJB-контейнером.
29. Что такое POJO?
30. Каким образом осуществляется конфигурирование EJB-компонента?

Глава 5

СЕРВИСНО-ОРИЕНТИРОВАННЫЕ ТЕХНОЛОГИИ РЕАЛИЗАЦИИ ИНФОРМАЦИОННЫХ СИСТЕМ

5.1. Сервисно-ориентированные архитектуры (СОА) и Web-сервисы

Сервисно-ориентированные архитектуры. Сервисно-ориентированная архитектура СОА (service-oriented architecture, SOA) — это подход к созданию ИС, основанный на использовании сервисов или служб (service). Далее сервис и служба рассматриваются как синонимы. СОА — это, в первую очередь, интеграционная архитектура, использование которой позволяет обеспечить гибкую интеграцию ИС. При использовании СОА приложения взаимодействуют, вызывая сервисы, входящие в состав других приложений. Сервисы объединяются в более крупные последовательности, реализуя бизнес-процессы, которые могут быть доступны как сервисы.

СОА можно рассматривать так же как подход к построению слабосвязанных (loosely coupled) систем, реализующих механизмы асинхронного взаимодействия. К слабосвязанным системам обычно относятся такие системы, как электронная почта и системы очередей сообщений.

Переход на СОА-архитектуры позволяет решать следующие задачи:

- уменьшать сроки освоения и внедрения новых ИТ-систем, быстро создавать новые ИТ-системы на базе уже существующих;
- уменьшать суммарную стоимость владения ИТ-продуктом и стоимость их интеграции;
- увеличить срок жизни ИТ-систем за счет возможности их оперативной модернизации;
- использовать гибкие модели ценообразования путем передачи разработки детализированных бизнес-модулей сторонним производителям (аутсорсинг);
- уменьшать стоимости работ по интеграции, необходимой при слиянии и поглощении компаний;
- реализовывать бизнес-процессы на уровне, не зависящем от приложений и платформ для поддержки процессов.

СОА — это интеграционная архитектура, основанная на концепции сервисов (служб).

Бизнес-функции и инфраструктурные функции, которые необходимы для построения распределенных систем, реализуются как сервисы, которые в сочетании или по отдельности обеспечивают прикладную функциональность либо другим сервисам, либо приложениям, взаимодействующим с конечным пользователем.

Концепция СОА предполагает использование единого механизма взаимодействия служб. Этот механизм строится на основе концепции свободных связей и должен поддерживать использование формальных интерфейсов.

СОА привносит преимущества, которые дают слабосвязанность и инкапсуляцию, в интеграцию на уровне предприятия.

Сервисы можно рассматривать как строительные блоки, которые могут использоваться для построения как сервисов более высокого уровня, так и законченных распределенных ИТ-систем. Сервисы могут вызываться независимо внешними или внутренними потребителями для выполнения элементарных функций либо могут объединяться в цепочки для формирования более сложных функций и для быстрого создания новых функций.

При использовании СОА организации могут создавать гибкие КИС, которые позволяют оперативно реализовывать быстро изменяющиеся бизнес-процессы и многократно использовать одни и те же компоненты в рамках одной ИТ-системы, в рамках семейств продуктов и в независимых ИТ-системах.

Сервисом можно назвать любую дискретную функцию, которая может быть предложена внешнему потребителю. В качестве сервиса может выступать как отдельная бизнес-функция, так и набор функций, которые образуют бизнес-процесс.

Сервисы, ориентированные на использование в составе СОА, должны обладать следующими свойствами:

- представлять собой многократно используемые бизнес-функции;
- определяться с помощью формальных, не зависящих от реализации интерфейсов;
- наличие протоколов связи, обеспечивающих прозрачность местонахождения и инвариантность по отношению к языку и платформе.

Хотя в качестве сервиса может выступать любая бизнес-функция, однако крайне желательно, чтобы имелась возможность повторного использования бизнес-функций одним или разными приложениями. Примером может служить система записи сообщений о событиях в log-файлы, которая присутствует практически в каждом приложении.

Во время выполнения каждый сервис размещается в одном, и только в одном месте и удаленно вызывается всеми клиентами, которые используют данный сервис. Достоинство подобного подхода состоит в том, что изменения в интерфейс или реализацию сервиса,

например изменение кода реализации или настроек, нужно вносить только в одном месте.

На этапе размещения каждая служба создается один раз, но ее можно локально размещать в любой системе или наборе систем, которым она необходима. Использование такого подхода позволяет повысить гибкость в достижении нужной производительности, а также гибкость настройки службы.

Использование формальных, не зависящих от реализации интерфейсов для определения и инкапсуляции функции службы особенно важно для того, чтобы службу можно было использовать многократно.

Интерфейс должен инкапсулировать только те аспекты поведения, которые используются при взаимодействии между клиентом и сервером. Формальное определение интерфейса или контракт служит для установления связи между поставщиком и потребителем службы. В нем должны быть указаны только те действия с обеих сторон, которые необходимы для осуществления взаимосвязи.

СОА не принуждает пользователя использовать конкретный протокол для доступа к сервису. Идея заключается в том, что сервис не определяется используемым протоколом, напротив, описание сервиса составляется независимым от протокола способом, позволяющим использовать для доступа к сервису разные протоколы. В идеале служба определяется только один раз при помощи интерфейса службы и должна иметь несколько реализаций для работы с разными протоколами доступа. Такой подход позволяет максимально расширить возможности повторного использования сервиса [32, 33 34].

Web-сервисы. В самом общем виде понятие Web-сервиса можно определить как сервис (услугу), которая предоставляется через WWW с использованием языка XML и протокола HTTP.

Практически все ведущие ИТ-компании положительно относятся к использованию Web-сервисов, поэтому Web-сервисы можно использовать в качестве механизма интеграции приложений, реализованных на любых платформах. Существует много разных определений понятия Web-сервиса.

В качестве «официального» определения можно рассматривать определение, которое дается консорциумом W3C:

«Web-сервис представляет собой приложение, которое идентифицируется строкой URI. Интерфейсы и привязки данного приложения описываются и обнаруживаются с использованием XML-средств. Приложения взаимодействуют посредством обмена сообщениями, которые пересылаются с использованием интернет-протоколов».

Иногда вместо термина Web-сервисы используется термин Web-услуги, в данной книге эти термины рассматриваются как синонимы.

Web-сервис — это новая парадигма реализации сервисов через Web. Иногда говорят о Web-сервис как об атрибуте Web третьего поколения.

При этом к первому поколению относят статический Web, использующий преимущественно статический HTML, а ко второму — интерактивный Web, использующий такие технологии, как PERL, ASP, JSP, и компоненты, используемые для построения распределенных приложений, которые работают по принципу черного ящика.

Web-сервис в полной мере и практически без всяких ограничений поддерживает кроссплатформенность, независимость от языка программирования, хорошо работает через файерволы.

Для обмена данными Web-сервис использует такие протоколы как XML, HTTP, TCP/IP, т.е. протоколы, которые поддерживаются практически всеми производителями.

Web-сервис оказал существенное влияние на подходы к построению распределенных систем, поскольку они обладают такими ценными свойствами, как самоописываемость, их легко создавать размещать, регистрировать и находить в репозитариях. Для того чтобы работать с сервисами, достаточно знать их интерфейсы. Унаследованные приложения могут быть относительно легко оформлены в виде Web-сервисов.

Появление данной технологии дало существенный толчок в развитии таких технологий, как e-Commerce, B2B, Enterprise Application Integration (EAI).

Сервисно-ориентированная парадигма программирования имеет много общего с компонентно-ориентированной парадигмой. Некоторые авторы [59] рассматривают Web-сервис как одну из разновидностей компонентов. Это касается возможности повторного использования сервисов и наличия некоторой «склеивающей» среды, которая используется в качестве инструмента для создания приложений на базе Web-сервиса.

Уже в начале XXI в. появились тысячи общедоступных сервисов. Большое число производителей, включая всех основных производителей ИТ-систем, предлагают инструментальные средства для разработки Web-сервисов. К наиболее успешно используемым можно отнести следующие инструментальные средства для создания Web-сервис: Apache Tomcat Next Generation Web service — AXIS, Microsoft.NET Web service studio based on IIS server, IBM Web Sphere Web service, BEA Web Logic Workshop, Java Web Service Development Pack (JWSDP), Mind Electric GLUE и многие другие. Значительное количество из них — свободно распространяемые. Web-сервисы, в частности, можно создавать с помощью таких средств разработки, как NetBeans и Eclipse.

Web-сервисы представляют собой самостоятельные модульные приложения, которые могут быть описаны, опубликованы, размещены и вызваны как локально, так и удаленно. Web-сервисы могут инкапсулировать как простейшие бизнес-функции типа «запрос-ответ», до полномасштабных взаимодействий бизнес-процессов. Службы могут создаваться заново или строиться на основе существующих приложений методом «обертывания».

Свойства Web-сервисов. Все Web-сервисы обладают следующими свойствами:

- являются самодостаточными, т. е. с клиентской стороны не требуется никакого дополнительного программного обеспечения кроме языка программирования, поддерживающего работу с XML и HTTP, а на серверной стороне требуется только HTTP-сервер, поддерживающий работу с посланиями;
- являются самоописываемыми, поскольку метаданные передаются вместе с сообщением и не требуется никаких внешних хранилищ метаданных;
- могут быть опубликованы, обнаружены и вызваны через Интернет. Причем для этого используют простые установившиеся стандарты, такие как HTTP и существующую сетевую инфраструктуру;
- являются модульными, т. е. простые Web-сервисы могут объединяться в более сложные, причем это может быть сделано разными способами, например, с использованием рабочих процессов или через прямой вызов Web-сервисов из других бизнес-процессов;
- инвариантны к способу реализации, т. е. клиент и сервер могут быть реализованы в разных средах с использованием разных языков программирования, причем для клиента не имеет значения, на какой платформе реализован сервер, и наоборот;
- открыты и основаны на стандартах, технической основой Web-сервисов являются XML и HTTP, значительная часть технологии Web-сервисов создана с использованием проектов с открытым исходным кодом;
- имеют свободные связи, по сравнению с другими, в частности, компонентными технологиями для Web-сервисов требуется более простой уровень координации, который позволяет осуществлять достаточно гибкую реконфигурацию для обеспечения интеграции нужных сервисов;
- являются динамическими, поскольку имеется возможность обнаруживать службы в процессе функционирования, при этом имеется возможность их модификации, которая не влияет на работу клиентов;
- являются действенным средством интеграции унаследованных приложений.

Различие между СОА и Web-сервисами состоит в том, что СОА представляет собой общую архитектуру интеграции приложений, а Web-сервисы — один из методов реализации СОА. Между Web-сервисами и СОА существует много логических связей. Web-сервисы представляют собой модель на основе открытых стандартов, которая может быть обработана автоматически и позволяет создавать формальные описания, не зависящие от интерфейсов служб. Web-сервисы предоставляют механизмы коммуникации, обеспечивающие прозрачность местоположения и возможность взаимодействия [6].

UDDI, ebXML
WSDL
SOAP, XML-RPC
HTTP, JMS, FTP, SMTP

Рис. 5.1. Стек протоколов, используемых для работы с Web-сервисами

Web-сервисы (Web Services) являются одной из реализаций сервисно-ориентированного подхода.

Наиболее важными компонентами архитектуры Web-сервисом являются провайдер сервиса (север) и пользователь сервиса (клиент).

Провайдер должен опубликовать (зарегистрировать сервис в репозитарии, который может быть реализован, в частности, как UDDI-реестр (Universal Description, Discovery, and Integration (UDDI) registry). UDDI-реестр внешне выглядит как Web-сервис и в нем хранится информация о зарегистрированных сервисах.

Клиент может обращаться к UDDI-реестру с запросами о месте нахождения отдельных сервисов и способах обращения к ним. Следует отметить, что для обращения к Web-сервисам клиент не обязательно должен предварительно обращаться к тому или иному репозитарию. Если клиенту известны местонахождение сервиса и его интерфейс, он может обращаться к сервису напрямую. Множество протоколов, используемых для работы с Web-сервисами, составляют стек (рис. 5.1).

На нижнем уровне находятся транспортные протоколы, отвечающие за транспортировку сообщений (HTTP, JMS, FTP, SMTP). Протоколы, определяющие форматы сообщений (SOAP, XML-RPC), располагаются на следующем уровне. На третьем уровне располагается протокол WSDL (Web Services Description Language, язык описания Web-служб). На четвертом уровне находятся протоколы, определяющие механизмы обнаружения Web-сервисов: UDDI, ebXML. Кроме перечисленных, имеется большое число вспомогательных протоколов, которые часто называют WS*. Все упомянутые протоколы [37, 38] будут рассмотрены далее.

5.2. Язык XML при работе с Web-сервисами

Основные понятия XML. Язык XML (eXtensible Markup Language) является основным при работе с Web-сервисами. Ниже дается краткое введение в XML. Более подробную информацию об этом языке можно найти во многих книгах, посвященных XML [19].

XML можно рассматривать как надмножество HTML. В XML пользователь может вводить и использовать собственные теги.

Любой XML-документ регламентируется метаданными, которые находятся в специальном файле.

Примеры метаданных: можно указать, что в году ровно 12 месяцев, в месяце может быть от 28 дней до 31 дня, в почтовом индексе должно быть ровно 6 цифр и т.д.

Для представления метаданных используется либо Document Type Definition (DTD), либо XSD файла. DTD является устаревшим. В 2001 г. консорциум W3C (WWW Consortium) рекомендовал описывать структуру документов XML на языке описания схем XSD.

Основным достоинством XML-файлов является то, что имеется возможность их машинной обработки. Практически все современные языки программирования имеют средства для работы с XML. XML используется в качестве универсального формата обмена данными, в качестве средства для хранения данных (XML базы данных), а также в разного рода конфигурационных файлах и файлах размещения (deployment descriptors).

Рассмотрим в качестве примера простейший XML-документ, содержащий записи о студентах и их рейтинге (среднем балле):

```
<?xml version="1.0" encoding="Windows-1251"?>
<students>
         <student id=123405>
              <name>Иванова Ю.А.</name>
              <rating>4.5</rating>
         </student>
              <student id=123410>
              <name>Соколов М.М.</name>
         <rating>4.3</rating>
</student>
    </students>
```

Документ XML начинается с пролога, в котором указывается версия языка XML, способ кодировки (по умолчанию — UTF-8) и ряд других элементов.

Все элементы XML-документа находятся внутри корневого элемента (root element), в данном случае — это элемент <students>. Имя корневого элемента должно совпадать с именем документа. Внутри корневого элемента имеются два вложенных элемента типа <student>. Каждый студент имеет уникальный идентификатор (id), в качестве которого может выступать, например, номер группы или номер студента по списку. В рассматриваемом примере идентификатор помещен в качестве атрибута в открывающийся тег. Внутри каждого элемента типа <student> имеется два вложенных элемента: <name> и <rating>. В первом указаны фамилия и инициалы, а во втором — средний балл.

Для приведенного выше примера XSD Schema выглядит следующим образом:

```
<?xml version="1.0" encoding=" Windows-1251"?>
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
         targetNamespace=http://www.myuniver.ru"
         xmlns="http://www.myuniver.ru">
    <xsd:element name = "students">
```

```
          <xsd:complexType>
          <xsd:sequence>
               <xsd:element ref="student"
               minOccurs="1"
               maxOccurs="unbounded"/>
          </xsd:sequence/>
          </xsd:complexType/>
</xsd:element>
     <xsd:element name="student">
     <xsd:complexType>
     <xsd:sequence>
     <xsd:element ref="name"/>
     <xsd:element ref="rating"/>
     </xsd:sequence>
     </xsd:complexType>
     </xsd:element>
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="rating" type="xsd:float"/>
</xsd:schema>
```

Использование пространств имен (namespace) является средством предотвращения коллизии имен. Для этого имена тегов и атрибутов снабжают префиксом, который отделяется от имени двоеточием.

Префикс имени связывается с идентификатором, определяющим пространство имен. Все имена тегов и атрибутов, префиксы которых связаны с одним и тем же идентификатором, образуют одно пространство имен. Идентификатор пространства имен имеет форму URI, причем такой адрес, как http://www.myuniver.ru/2011/mns, может не соответствовать никакому реальному адресу, а представлять собой просто уникальный идентификатор. Каждый пользователь может создавать любое количество собственных пространств имен и пользоваться общедоступными пространствами имен. В рассматриваемом примере в качестве разделяемого пространства имен выступает xsd:schema, которому соответствует пространство имен xmlns:xsd=http://www.w3.org/2001/XMLSchema.

В данном пространстве имен определены теги, относящиеся к XSD Schema.

Появление имени тега без префикса в документе, использующем пространство имен, означает, что имя принадлежит пространству имен по умолчанию (default namespace).

В рамках схемы можно определять новые типы данных, например:

```
<element name = "student"
<complexType>
     <element name = "name" type = "xsd: string" />
     <element name = "rating" type = "xsd: float" />
```

```
    </complexType>
</element>
```

В данном случае определяется новый комплексный тип.

Протокол XML-RPC. Непосредственным предшественником Web-сервисов являлся протокол XML-RPC.

Это очень простой протокол, предназначенный для вызова удаленных процедур. В отличие от традиционного RPC для вызова удаленной процедуры используются XML-послания. Ответ приходит также в форме XML-послания.

Рассмотрим простейший пример.

Пусть имеется удаленная процедура, осуществляющая поиск синонимов в словаре. Для обращения к процедуре используется строка вида public String getSynonym (String word). В качестве аргумента используется исходное слово (word). Процедура возвращает слово — синоним исходного.

Запрос в рассматриваемом случае выглядит следующим образом:

```
<?xml version="1.0"?>
<methodCall>
    <methodName>getSynonym</methodName>
    <params>
        <param>
            <value>
                <string>самолет</string>
            </value>
        </param>
    </params>
</methodCall>
```

Корневым является элемент <methodCall>, в который вложен элемент <methodName>, в теле которого записывается имя вызываемой процедуры. Параметры вызываемой процедуры помещаются в элемент <params>. В элемент <params> вложены нуль или несколько элементов <param>, в которые помещаются параметры вызываемой процедуры.

В рассматриваемом примере требуется получить синоним слова «самолет».

Ответ также поступает в форме XML-сообщения:

```
<?xml version="1.0"?>
<methodResponse>
    <params>
        <param>
            <value>аэроплан</value>
        </param>
    </params>
</methodResponse>
```

Если произошла ошибка при выполнении запроса, то в ответном пакете указывается код ошибки. XML-RPC позволяет использовать такие типы данных как целые строки, вещественные, структуры, массивы.

Механизм XML-RPC был создан в середине 1990-х гг., однако не получил широкого распространения. Причины этого достаточно понятны и состоят в следующем:

• запросы, выполняемые в рамках XML-RPC, только отчасти являются самоописываемыми, поскольку пространства имен в нем не определены;

• XML-RPC подобно классическому RPC не поддерживают работу с объектами;

• при работе с XML-RPC обнаруживаются проблемы, вызванные использованием XML. На стороне клиента необходимо сформировать запрос. На стороне сервера необходимо его разобрать. То же самое требуется сделать с ответом. Все это усложняет код и требует существенных временных затрат;

• сообщения, отправляемые в формате XML, за счет тегов имеют большую избыточность.

Несмотря на отмеченные недостатки, данная технология получила дальнейшее развитие [19].

Протокол SOAP. SOAP — это основанный на XML протокол, определяющий механизм обмена сообщениями. Протокол SOAP появился в 1998 г. как W3C спецификация. В ранних версиях спецификации SOAP расшифровывался как Simple Object Access Protocol. В последних версиях этот термин никак не расшифровывается. На момент написания данной книги последней была версия 1.2 от апреля 2007 г. (спецификация находится по адресу http://www.w3.org/TR/2007/).

Хотя SOAP имеет много общего с XML-RPC, но имеются и принципиальные отличия от него.

Во-первых, протокол SOAP не различает вызов процедуры и ответ на него, а просто определяет формат послания (message) в виде документа XML, который может содержать вызов процедуры, ответ на него, запрос на выполнение каких-то других действий или просто текст.

Во вторых, SOAP-послание представляет собой самоописываемый документ, включающий, в частности, описания пространств имен. Структуру SOAP-послания определяет соответствующая XML Schema. Для пересылки SOAP-посланий обычно используется метод HTTP POST, хотя можно использовать и другие протоколы, такие как FTP, SMTP.

SOAP-послание включает два элемента: заголовок (Header) и тело (Body), которые помещаются в контейнер (Envelope). SOAP-послание представляет собой правильный XML-документ. Заголовок является факультативным элементом, а тело — обязательным.

В качестве примера рассмотрим пример сервера, который находит синоним слова с посредством вызова метода getSynonym:

```
<?xml version="1.0" encoding="Windows-1251"?>
<SOAP-ENV:Envelope
SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAPENV="
    http://schemas.xmlsoa.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/
      XMLSchema-instance"
    xmlns:SOAPENC="
    http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
    <getSynonym>
    <word xsi:type="xsd:string">аэроплан</word>
    </getSynonym>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

В теле запроса в элемент word, который имеет тип XSD string, помещается исходное слово. В ответ на запрос клиенту поступает ответное послание, которое имеет вид:

```
<?xml version="1.0" encoding="Windows-1251"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
      envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/
      XMLSchema-instance">
    <SOAP-ENV:Body>
        <getSynonymResponse
        SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">
        <getSynonymResult xsi:type=
            "xsd:string">самолет</getSynonymResult>
        </getSynonymResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Рассмотрим пример создания простейшей службы, которая может находить синоним.

Имеется много разных систем поддержки работы с Web-сервисами. Одной из популярных является Apache eXtensible Interaction System (AXIS).

Создание серверной части не вызывает проблем. Для работы с AXIS достаточно написать текст сервиса. Для этого создаем файл SynonymService.java:

```
// SynonymService.java
public class SynonymService {
    public String getSynonym(String word) {

        //Находим в словаре требуемый синоним
        ...............
    //Возвращаем синоним
    return "" + synonym;
}
}
```

Данный файл необходимо переименовать в SynonymService.jws и не компилируя поместить в каталог webapps (при работе с AXIS). После этого сервис готов к использованию

Клиентская часть приложения сложнее. При работе с AXIS вначале создается объект класса service, который предназначен для установления связи с Web-сервисом. Он предоставляет экземпляр класса call, в который заносятся параметры запроса, в частности, адрес и имя Web-сервиса «getSynonym». После того как запрос сформирован, Axis обращается к Web-сервису посредством вызова метода invoke (). Запрос направляется серверу приложений, работающему по указанному в запросе адресу. На сервере запускается сервлет AxisServiet, разбирающий SOAP-послание. Он находит требуемый .jws-файл, компилирует и запускает с требуемыми параметрами. После выполнения требуемых действий сервлет формирует SOAP-послание, в которое помещается результат. Послание отправляется клиенту. Там оно разбирается. Для клиента результат доступен как результат выполнения метода invoke () [36].

5.3. WSDL-описание

WSDL используется для описания интерфейсов Web-сервисов. Средствами WSDL можно описать, где находится Web-сервис и каким образом к нему следует обращаться. WSDL-описание позволяет создавать независимое от платформы и языка реализации описание Web-сервиса. Нетрудно заметить, что WSDL имеет много общего с IDL, используемым в RPC. WSDL-описание представляет собой XML-документ, структура которого показана на рис. 5.2.

В качестве корневого элемент WSDL-описания используется элемент <definitions>, в который вкладываются семь элементов, пять из которых являются основными, а два — вспомогательными.

В качестве основных выступают элементы <types>; <message>; <portType>; <binding>; <service>.

В качестве вспомогательных выступают элементы <import>; <documentation>.

<definitions>

<types>	Что делается?
<message>	
<portType>	
<binding>	Как делается?
<service>	Где находится?
<import>	
<documentation>	

</definitions>

Рис. 5.2. Структура XML-документа

Каждый элемент имеет собственное имя, определяемое обязательным атрибутом name. Имена используются для ссылки на отдельные элементы.

Элемент <types> присутствует, если требуется определить сложные типы с помощью языка XSD. В противном случае данный элемент отсутствует.

Элемент <message> описывает каждое из SOAP посланий в терминах «запрос-ответ». В этот элемент вкладываются элементы <part>. При использовании процедурного стиля в каждый такой элемент описывается имя и тип одного аргумента запроса или возвращаемого значения. При использовании документного стиля элементы <part> могут описывать отдельную часть послания MIME-типа «multipart/related».

Элемент <portType> описывает интерфейс Web-сервиса, который также называют *пунктом назначения* (endpoint) или *портом* (port) сервисов, называемых *операциями*. Отдельная операция описывается как вложенный элемент <operation>, который описывает отдельный сервис. Имеются четыре вида сервисов: два простых действия (получение послания и отправка ответа) и два комбинированных действия (отправка послания — получение ответа и получение послания — отправка ответа). Действия типа получение и отправка посланий описываются вложенными элементами <input> и <output>, а также сообщением об ошибке (элемент <fault>). Элемент <serviceType> содержит множество вложенных элементов <portType>. Один и тот же порт может быть связан с несколькими сервисами.

Элемент <binding> описывает формат пересылки послания: протоколы и его тип. Каждый элемент <message> может быть связан с несколькими такими элементами<binding>, по одному для каждого способа пересылки.

Элемент <service> определяет местонахождение сервиса как один или несколько портов, каждый из которых описывается вложенным элементом <port>, содержащим адрес интерфейса Web-сервисы.

Элемент <import> включает файл с XSD схемой описания WSDL или другой WSDL-файл.

Элемент <documentation> представляет собой комментарий, который можно включить в любой элемент описания WSDL.

Принято говорить, что элементы <types>, <message> и <portType> определяют, какие именно услуги предоставляет данный сервис.

Элементы <binding> определяет, как реализована Web-служба, т. е. как к ней обращаться.

Элементы <service> определяет, где располагается сервис.

Ниже в качестве примера приведен фрагмент WSDL-описания.

```
<?xml version="1.0" encoding="UTF-8"?>
name="SynonymService"
targetNamespace="http://www.vocab.ru/Thesaurus.wsdl"
xmlns="http://www.w3.org/ns/wsdl"
........
<wsdl:definitions...
     <wsdl:message name="getSynonymResponse">
          <wsdl:part name="synonym" type=
             "SOAP-ENC:string"/>
     </wsdl:message>
     <wsdl:message name="getSynonymRequest">
          <wsdl:part name="word" type=
             "SOAP-ENC:string"/>
     </wsdl:message>
     <wsdl:portType name="Synonyms">
          <wsdl:operation name="getSynonym"
             parameterOrder="word">
               <wsdl:input message="intf:
                  getSynonymRequest"/>
               <wsdl:output message="intf:
                  getSynonymResponse"/>
          </wsdl:operation>
     </wsdl:portType>
     <wsdl:binding name="getSynonymSoapBinding"
        type="intf:Thesaurus">
          <wsdlsoap:binding style="rpc"
          transport="http://schemas.xmlsoap.org/soap/
             http"/>
          <wsdl:operation name="getSynonym">
          <wsdlsoap:operation soapAction=""/>
               <wsdl:input>
                    <wsdlsoap:body
                    encodingStyle="http://schemas.
                       xmlsoap.org/soap/encoding/"
                    namespace="urn:myDirectory"
```

```
                use="encoded"/>
          </wsdl:input>
          <wsdl:output>
               <wsdlsoap:body
               encodingStyle=
               "http://schemas.xmlsoap.org/soap/
                 encoding/"
               namespace="urn:myDirectory"
                 use="encoded"/>
          </wsdl:output>
     </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="SynonymService">
  <wsdl:port binding="intf: getSynonymSoapBinding"
    name="Thesaurus">
      <wsdlsoap:address
          location="http://localhost:8080/axis/
            services/Thesaurus"/>
      </wsdl:port>
  </wsdl:service>
<documentation> Thesaurus WSDL File</documentation>
</wsdl:definitions>
```

Предполагается, что данное описание находится в файле Thesaurus. wsdl.

Имя сервиса SynonymService определено в элементе service.

Далее определяются используемые пространства имен (часть определений пропущена).

В элементах <wsdl:message> описаны используемые типы SOAP посланий. Используются два типа сообщений, которые называются getSynonymRespons и getSynonymReques. Первое содержит поле synonym, а второе — поле word. Оба поля представляют собой строки символов.

Элемент <wsdl:portType> называется Synonyms и выполняет только одну операцию getSynonym, которая получает входную переменную word. Входное послание имеет имя getSynonymRequest, а выходное — getSynonymResponse.

Элемент <wsdl:binding»> называется getSynonymSoapBinding. В нем указывается, что сервис работает в режиме запрос-ответ (использует rpc-стиль), использует в качестве транспорта протокол HTTP, работает с SOAP-посланиями и выполняет операцию getSynonym. Затем в терминах SOAP повторяется описание операции getSynonym.

В элементе <wsdl:service>, который называется SynonymService, указывается местонахождение сервиса.

Элемент <documentation> может содержать любые комментарии.

Поскольку в рассматриваемом примере не используются сложные типы, то элемент <types> отсутствует.

WSDL-спецификация может быть получена автоматически из файла реализации Web-сервиса, например, из Java-файла или из описания интерфейса. Имеется много разных систем поддержки работы с Web-сервисом. Одной из наиболее популярных является Apache eXtensible Interaction System (AXIS).

Можно выделить два альтернативных подхода к разработке Web-сервисов: подход по принципу «сверху-вниз» и подход «снизу-вверх».

В первом случае разработка начинается с разработки WSDL-описания, которое используется для генерации скелетона и стаба, затем к ним добавляется бизнес-логика.

Во втором случае разработка начинается с создания кода для генерации WSDL-описания.

Например, в рамках существующих инструментальных средств имеются утилиты для выполнения упомянутых выше преобразований [19, 59].

5.4. UDDI-реестр

Реестр UDDI представляет собой техническую спецификацию для построения распределенных репозитариев, которые позволяют отдельным организациям опуликовывать и находить требуемые сервисы. Если клиенту известно местонахождение сервиса и способ работы с ним, то надобность в реестре отпадает. Реестр UDDI (UDDI Business Registry) — это распределенная база данных, и по принципу работы она подобна системе DNS. Для пользователя реестр UDDI доступен как Web-сервис.

Спецификация UDDI была разработана в 2000 г. фирмами IBM, Microsoft и Ariba. (На момент написания данной книги последняя версия 3.x (http://uddi.org/pubs/uddi_v3.htm).) В начале 2000-х гг. многие крупные компании организовали и содержали открытые (public) UDDI-реестры, где каждый желающий мог зарегистрировать собственный сервис и найти нужный сервис.

UDDI представляет собой документ в формате XML и имеет трехуровневую организацию.

На верхнем уровне UDDI-реестр представляет собой справочник, работающий по принципу *Белых страниц*, которые содержат базовую информацию о бизнесе, включая название, описание и контактную информацию (телефон, факс, Web-сайт и т. п.) и бизнес-идентификатор.

На втором уровне UDDI-реестр представляет собой *Желтые страницы*, в которых бизнес информация классифицируется по типу бизнеса, производимым продуктам. На третьем уровне UDDI-реестр

можно рассматривать как *Зеленые страницы*, где описаны способы доступа к сервисам.

Состав реестра UDDI. Реестр UDDI включает в себя основные и дополнительные элементы:

- основные элементы — бизнес-сущности, бизнес-сервисы, модель привязки, техническая модель сервиса;
- дополнительные элементы — утверждения, информация, подписка.

Элемент бизнес-сущность <businessEntity> содержит уникальный в пределах реестра ключ UUID (Unique Universal Identifier), название фирмы (вложенный элемент <name>), описание сферы деятельности организации, типы предоставляемых сервисов, контактную информацию, ссылки URL. Небольшой организации обычно соответствует одна бизнес-сущность, а большой организации с большим количеством подразделений может соответствовать несколько бизнес-сущностей.

Элементы бизнес-сервисы <businessServices> вкладываются в элемент бизнес-сущность и описывают каждый из предоставляемых сервисов. В состав описания входят ключ UUID-сервиса (serviceKey), имя сервиса (вложенный элемент <name>), описание и (или) ссылки на более подробную информацию. (Предоставляемые сервисы — это не обязательно Web-сервисы.)

Элементы модели привязки <bindingTempiates> вкладываются в элемент <businesService> и описывают конкретные способы работы с данным сервисом. Это могут быть либо прямые ссылки в форме URL, либо косвенные, например, в форме WSDL-описания. Каждый элемент типа <bindingTempiate> имеет уникальный ключ UUID-указателя и содержит ссылку на соответствующий ему элемент <tModel>.

Элемент техническая модель сервиса <tModel> (technical Model) представляет собой подробное формальное описание каждой услуги. Данный элемент используется программным обеспечением и обычно оформляется как отдельный документ XML.

Кроме перечисленных выше, в UDDI-реестре имеются дополнительные элементы.

Элемент утверждение <publisherAssertion> представляет собой описание отношений между фирмами. Примерами отношений могут служить «parent-child»), т.е. одна организация является подразделением другой. Утверждение типа «identity»означает, что речь идет об одной и той же организации.

Элемент информация <operationalInfo> содержит даты создания и последней модификации записи в реестре, идентификатор узла реестра, идентификатор владельца информации.

Элемент подписка <subscription> содержит данные, которые используются для информирования пользователей о тех или иных системных событиях.

Ниже приведено описание структуры UDDI-реестра:

```
<businessDetail>
<businessEntity businessKey=
     "[Уникальный идентификатор]">
Имена, описания, контакты,..
<businessServices>
<businessService serviceKey=
     "[Уникальный идентификатор]"
     businessKey="[Уникальный идентификатор]">
Имена, описания,..
<bindingTemplates>
<bindingTemplate bindingKey=
     "[Уникальный идентификатор]" serviceKey=
     "[Уникальный идентификатор]">
описания..
<accessPoint
URLType="http">http://www … </accessPoint>
<tModelInstanceDetails>
<tModelInstanceInfo>
<tModelKey>[Уникальный идентификатор]</tModelKey>
Descriptions,..
<instanceDetails>
Описания, URL,..
</instanceDetails>
</tModelInstanceInfo>
</tModelInstanceDetails>
</bindingTemplate>
…
</bindingTemplates>
<categoryBag> … <tModelKey>
     [Уникальный идентификатор]</tModelKey> …
</categoryBag>
</businessService>
…
</businessServices>
<identifierBag> … <tModelKey>
     [Уникальный идентификатор]</tModelKey> …
</identifierBag>
<categoryBag> … <tModelKey>
     [Уникальный идентификатор]</tModelKey> …
     </categoryBag>
</businessEntity>
…
</businessDetail>
```

Программный интерфейс UDDI. Для работы с реестром UDDI пользователю предоставляется API.

Функции, входящие в состав UDDI API, можно разделить на четыре группы:

• функции, позволяющие регистрировать и изменять описания сервисов в UDDI реестре, которые называют save_xxx функции;

• функции для получения информации о сервисах, которые называют get_xxx функции;

• функции для поиска сервисов, которые называют find_xxx функции;

• фупкции для удаления сервисов, которые называют delete_xxx функции;

Под символами «xxx» подразумевается тип объекта («business», «service», «binding», «tModel»).

С точки зрения пользователя, UDDI-реестр — это Web-сервис, для обращения к которому ему следует направить SOAP-послание, поэтому пользовательский интерфейс можно определить и в терминах SOAP-посланий, которые можно направлять на вход реестра.

Для работы с UDDI-реестром существует много библиотек классов и отдельных фупкций, написанных на разных языках. В среде программистов Java часто используется пакет классов UDDI4J (UDDI for Java), разработанный фирмой IBM, который доступен на сайте http://sourceforge.net/projects/uddi4j/. (На момент написания книги последней версией была версия 2.0.5 от 2006 г.)

В данном пакете все обращения к реестру UDDI идут через класс-посредник UDDIProxy, который преобразует все обращения в функции UDDI API. Методы этого класса называются так же, как и функции UDDI API: save_business (), find_service () и т.п. Они возвращают объекты классов, которые представляют собой SOAP-структуры [37, 38].

5.5. Бизнес-реестр ebXML

Электронный реестр для поддержки систем электронного бизнеса (ebXML) был разработан центром международной торговли и электронного бизнеса ООН (http://www.uncefact.org/) и (http://www.oasis-open.org/).

Информация о ebXML доступна на сайте http://www.ebxml.org/.

Основной целью разработки ebXML было создание протокола, описывающего взаимодействия партнеров по типу Business to Business (B2B).

В ebXML организации, осуществляющие взаимодействие, называются сторонами (parties). Стороны реализуют деловое сотрудничество (Business Collaboration). В реестре ebXML хранится информация о каждой стороне и их сотрудничестве. Информация о сотруд-

ничестве описывается в терминах заявлений о сотрудничестве CPP (Collaboration Protocol Profile) и соглашений о сотрудничестве CPA (Collaboration Protocol Agreement).

Соглашение о сотрудничестве — это также XML-документ, в котором находятся технические сведения о соглашениях двух сторон, которые собираются из заявлений о сотрудничестве или составляются стороной, заинтересованной в получении услуг.

Заявление о сотрудничестве формируются во время переговоров, в процессе которых стороны обмениваются сообщениями.

Соглашение достигается следующим образом.

1. Сторона, оказывающая услуги, составляет заявление о сотрудничестве и регистрирует его в реестре ebXML.

2. Сторона, заинтересованная в получении услуги, находит в реестре подходящее заявление о сотрудничестве.

3. Сторона, которая хочет получить услуги, создает собственное заявление, затем на основании обоих заявлений составляет проект соглашения и посылает его стороне, оказывающей услуги.

4. Стороны достигают соглашения, обмениваясь сообщениями. После достижения соглашения стороны хранят копии соглашения CPA на своих серверах и/или в реестре.

При обмене сообщениями может использоваться электронная подпись.

Более подробную информацию можно найти в [19], а также на официальном сайте ebXML (http://www.ebxml.org/).

5.6. Язык WS-Inspection для поиска Web-служб

Одним из самых старых и простых реестров является система WS-Inspection, которая была разработана фирмами Microsoft и IBM в 2001 г. Основной целью создания данного средства было предельное упрощение процедуры поиска сервиса. В отличие от UDDI и ebXML, которые решают задачи не только поиска, но также описания и изменения записей, WS-Inspection позволяет только найти требуемый сервис в пределах конкретного сервера. Если UDDI и ebXML — это распределенные базы данных, то для работы WS-Inspection требуется только один XML-файл и поддержка сервера приложений.

В рамках WS-Inspection задача поиска решается следующим образом. Web-сервисы описываются на языке WS-Inspection Language (WSIL) и помещаются в файл с расширением .wsil, который представляет собой XML-файл. Данный файл помещается в определенный каталог сервера приложений. Получив запрос, сервер приложений просматривает все wsil-файлы и находит описание требуемого сервиса. Файл wsil для словаря синонимов будет иметь следующую очень простую структуру:

```
<?xml version="1.0"?>
<inspection
xmlns="http://schemas.xmlsoap.org/ws/2001/10/
inspection/">
     <service>
     <name>SynonymService</name>
          <description
          referencedNamespace="http://schemas.xmlsoap.
            org/wsdl/"
          location="http://www.myvocab.ru/wsdl/
            Thesaurus.wsdl"/>
     </service>
</inspection>
```

Корневой элемент wsil — это элемент <inspection>, в него вкладывается единственный в рассматриваемом случае элемент <service>. Рассматриваемый вариант описания является простейшим. В элемент <service> вкладывается элемент <name>, содержащий название сервиса и элемент <description>, в котором содержится в месте нахождения .wsdl-описания.

После того как wsil-файл подготовлен и помещен в соответствующий каталог, клиент может к нему обращаться. В соответствии со спецификацией WS-Inspection в корневом каталоге сайта, поддерживающего работу с WS-Inspection, должен находиться файл с именем inspection.wsil.

Если на момент обращения такой файл отсутствует, то он создается автоматически.

Например, если обратиться к серверу с браузера со стороны клиента, например, http://www.myvocab.ru/inspection.wsil, то на экране появится список сервисов, реализуемых на данном сайте.

Для работы с WS-Inspection были созданы соответствующие инструментальные средства, в частности пакет IBM WSTK, включающий пакет интерфейсов и классов WSIL4J, для работы на языке Java.

При обращении к inspection.wsil со стороны программы-клиента сервер передает запрос сервлету. WSILServet, который просматривает все файлы с расширением wsil, отправляет их клиенту.

В состав пакета входит класс WSILProxy. Этот класс используется для формирования запроса. В него помещается также информация, полученная с сервера.

Более подробное описание работы с WS-Inspection можно найти в [19].

5.7. Спецификации WS-*

Помимо основных спецификаций и стандартов, таких как SOAP WSDL и UDDI, существует ряд спецификаций, предназначенных

для расширения концепции Web-служб на основе SOAP и WSDL в целях обеспечения надежности, безопасности, поддержки состояния и качества обслуживания. Указанные спецификации, названия большинства из которых начинаются с префикса WS-, основываются на технологиях W3C. Они ориентированы преимущественно на использование в рамках КИС, на решение частных задач в рамках СОА, таких как поддержка работы с транзакциями, маршрутизация, безопасность и работа с состояниями.

WS-Coordination и WS-Transaction. В большинстве своем транспортные протоколы Интернет не поддерживают работу с состоянием и являются ненадежными, поэтому их использование для реализации транзакции вызывает проблемы. Это приводит к тому, что если разработчик хочет реализовать транзакции в рамках бизнес-процессов, которые будут рассмотрены далее, то ему придется реализовывать механизмы работы с транзакциями внутри приложения.

Вызовы сервисов, осуществляемые через асинхронный обмен сообщениями, должны обладать возможностью пакетного выполнения в качестве единой атомарной единицы, чтобы при сбое хотя бы одного вызова выполнять откат всех вызовов как единого целого [17]. Спецификации WS-Coordination и WS-Transaction решают описанную проблему для Web-служб.

Спецификация WS-Coordination определяет способ создания и распространения информации о контексте всеми службами, участвующими в продвижении потока сообщений в асинхронном режиме или при наличии сбоев синхронизации.

WS-Transaction определяет способ мониторинга и измерения успешности или неуспешности отдельного каждого действия в потоке действий. Когда сообщение SOAP пребывает в конечную точку, из него извлекается заголовок и передается для анализа координатору транзакций.

В рамках спецификации WS-Transaction определяются два типа координации: атомарная транзакция (atomic transaction — AT) и бизнес-активность (business activity — BA).

Атомарная транзакция применяется для осуществления операций с относительно коротким временем жизни, в которых допускаются блокирование ресурсов, таких как нити, элементы данных и т. п.

Бизнес-активность представляет собой «долгоживущий» процесс, который может состоять из нескольких атомарных транзакций. Глобальный откат такой большой транзакции в случае возникновения единственного сбоя обычно нежелателен. Гораздо лучше, если сбой одной атомарной транзакции в рамках бизнес-активности будет инициировать запуск другого набора вызовов сервисов и отправку других сообщений. Отправка сообщений может представлять собой некоторый механизм, позволяющий устранить последствия ошибки таким образом, чтобы сохранить часть истории выполнения бизнес-активности.

WS-Reliability и WS-ReliableMessaging. Спецификация WS-Reliability определяет механизм обмена сообщениями в асинхронном режиме, с гарантированной доставкой, без дубликатов и с упорядочиванием сообщений.

Для реализации данного механизма к заголовкам SOAP посланий добавляются элементы MessageHeader, ReliableMessage, MessageOrder и RMResponse, которые содержат идентификаторы сообщений, такие как порядковый номер сообщения в последовательности сообщений и идентификатор группы, метку времени, время жизни, тип сообщения, сведения об отправителе и получателе, а также информацию о подтверждении получения.

Получатель послания либо подтверждает его получение, либо отправляет сообщение об ошибке, используя элемент RMResponse заголовка SOAP. Если отправитель исходного сообщения не получит подтверждение, он еще раз отправит то же самое сообщение с тем же идентификатором, что и у исходного сообщения. Отправитель обязан хранить исходное сообщение, пока у того не истечет время жизни, либо пока не будет получено подтверждение получения или же пока получатель не сообщит об ошибке. Получатель также обязан хранить сообщение, пока оно не будет надежно передано логике приложения.

Чтобы гарантировать доставку сообщений по принципу «один и только один раз», в спецификации WS-Reliability предусмотрен механизм хранения порядкового номера сообщения в последовательности сообщений. Сообщения SOAP, сгруппированные в последовательность, могут иметь общий идентификатор группы, однако в заголовке каждого из них будет указан собственный порядковый номер. Это поможет получателю расположить сообщения в правильном порядке, прежде чем передать их приложению.

В спецификации WS-ReliableMessaging описывается аналогичный протокол, предназначенный для надежной доставки сообщений между распределенными приложениями при наличии сбоев. Его описание не основывается на какой-либо конкретной технологии, а значит данный протокол может быть реализован с использованием целого ряда сетевых транспортных технологий и привязок.

Спецификация WS-ReliableMessaging опирается на те же принципы, что и WS-Reliability. В ней тоже используются подтверждения, обратные вызовы, идентификаторы и постоянные хранилища сообщений, а также предполагается выдача подробных сообщений об ошибках. Кроме того, она обеспечивает доставку сообщений в соответствии с одной из четырех базовых схем доставки: не более чем один раз; как минимум один раз; один и только один раз; в том же порядке, в котором были отправлены сообщения.

Ключевым различием между конкурирующими спецификациями надежности является тот факт, что WS-ReliableMessaging опирается на другие важные спецификации Web-служб, такие как WS-Security и WS-Addressing.

WS-Conversation и WS-Security. Спецификация WS-Conversation описывает протокол, предназначенный для управления асинхронным обменом сообщениями с поддержкой состояния между отправителем и получателем.

WS-Security является своеобразным мостом между SOAP и существующими технологиями безопасности и включает в себя описание универсальных, расширяемых средств сопоставления маркеров безопасности с сообщениями SOAP и передачи этих маркеров конечным точкам SOAP.

Она позволяет работать с двоичными маркерами безопасности, такими как цифровые сертификаты и билеты Kerberos. Данная спецификация определяет универсальный набор механизмов, позволяющих реализовать различные протоколы безопасности и внедрить в схему защиты любое число доменов доверительных отношений, форматов цифровых подписей и технологий шифрования.

Помимо проверки подлинности отправителя, WS-Security может обеспечивать целостность сообщения. Это выполняется посредством использования разработанных консорциумом W3C стандартов XML Signature и XML Encryption.

В модели безопасности, описанной в WS-Security, предполагается, что отправитель сообщения SOAP передает вместе с сообщением ряд сведений о себе: идентификатор отправителя, группу, привилегии и т. п. Все эти сведения группируются в так называемый подписанный маркер безопасности. Получатель сообщения должен подтвердить эти сведения. Маркеры и подписи передаются внутри заголовка SOAP, а точнее — с помощью элемента security пространства имен ws-Security.

Из всех спецификаций WS-* эта получила наибольшую поддержку среди главных производителей платформ для Web-служб.

WS-Addressing, WS-Policy и другие спецификации WS-*. Данные спецификации — частные аспекты Web-сервисов. WS-Addressing, определяющая XML-элементы, с помощью которых в сообщении идентифицируются конечные точки Web-служб, является стандартным способом задания, кем было отправлено сообщение («От:») и кому оно предназначается («Кому:»), что облегчает решение задачи рассылки посланий на списка получателей.

Спецификация WS-Policy определяет синтаксис описания политик Web-сервиса. Политики описывают требования к службе, предпочтения, возможности и желаемое качество обслуживания. Сопутствующая спецификация Web Services Policy Assertions Language (WS-PolicyAssertions) позволяет проверить, поддерживает ли конечная точка службы или сообщения конкретную политику. Процесс проверки включает в себя анализ объявлений WS-Policy на предмет конкретной требуемой политики.

Следует отметить, что перечисленные выше стандарты поддерживаются далеко не всеми производителями инструментальных средств для работы с Web-сервисами [61].

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Охарактеризуйте понятие системы, ориентированной на работу с сообщениями.
2. Что такое очереди сообщений?
3. Охарактеризуйте существующие модели обмена сообщениями: точка-точка и публикация-подписка.
4. Что такое JMS?
5. Охарактеризуйте понятие сервисно-ориентированная архитектура.
6. Охарактеризуйте понятие сервис.
7. Что такое Web-сервисы?
8. Что такое XML?
9. Каковы основные правила построения XML-документа?
10. Что такое DTD и XSD?
11. Что такое XML-RPC?
12. Что такое SOAP, какова структура SOAP-послания?
13. Что такое и каково назначение WSDL?
14. Какова структура WSDL-описания?
15. Что такое UDDI-реестр?
16. Какова структура UDDI-описания?
17. Что такое бизнес-реестр ebXML?
18. Что такое WS-*?

Глава 6
ИНТЕГРАЦИЯ ПРИЛОЖЕНИЙ

6.1. Общие принципы организации взаимодействий в информационных системах

Системы обмена данными подразделяют на системы, использующие асинхронную и синхронную связь, а также на системы, работающие по принципу сохранной и несохранной связи [22].

При использовании *асинхронной связи* (asynchronous communication) отправитель после отправки сообщения немедленно продолжает работу, а сообщение сохраняется в локальном буфере передающего хоста или на ближайшем коммуникационном сервере. При использовании *синхронной связи* (synchronous communication) работа отправителя блокируется до того момента, когда сообщение будет доставлено получателю, либо сохранено в локальном буфере принимающего хоста.

При использовании синхронной связи (synchronous communication) можно выделить три степени «жесткости»:

- отправитель может продолжать работу после того, как сообщение помещено во входной буфер получателя;
- работа отправителя блокируется до момента получения сообщения непосредственно пользователем, в этом случае от пользователя часто требуется подтвердить прием сообщения;
- работа отправителя блокируется до момента получения ответа.

При *сохранной связи* (persistent communication) сообщение, предназначенное для отсылки, хранится в коммуникационной системе до тех пор, пока его не удастся передать получателю. Сообщение сохраняется на коммуникационном сервере до тех пор, пока его не удастся передать на следующий коммуникационный сервер. Поэтому у приложения, отправляющего сообщение, нет необходимости после отправки сообщения продолжать работу. Приложение, принимающее сообщение, не обязательно должно находиться в рабочем состоянии во время отправки сообщения.

При использовании сохранной связи сообщения никогда не теряются и не пропадают.

Альтернативой использования механизмов сохранной связи является использование *несохранные связи* (transient communication. При

применении нерезидентной связи сообщение хранится в системе только в течение времени работы приложений, которые отправляют и принимают это сообщение.

Если коммуникационный сервер не имеет физической возможности передать сообщение следующему серверу или получателю, то сообщение уничтожается. Следует отметить, что все коммуникационные сервисы транспортного уровня поддерживают только нерезидентную связь.

На практике применяются различные комбинации этих типов взаимодействия.

Системы, основанные на вызове удаленных процедур или обращении к удаленным объектам, по большей части являются синхронными и реализуют несохранные связи. Хотя после того, как стало очевидно, что данные виды связи не всегда самые подходящие, были созданы средства для менее жестких форм нерезидентной синхронной связи, таких как асинхронные вызовы RPC и отложенные синхронные операции.

Значительная часть существующих приложений представляет собой распределенные приложения, причем очень часто речь идет об интеграции уже существующих подсистем в единую ИС.

Типовыми проблемами, возникающими при создании распределенных ИС и, в частности КИС, являются следующие:

- различия между приложениями;
- необходимость внесения изменений в код интегрируемых приложений;
- ограниченная скорость передачи данных;
- ненадежность сетевой инфраструктуры.

Отдельные приложения, входящие в состав распределенной системы, могут работать на разных аппаратных и программных платформах; написаны могут быть на разных языках программирования, использовать разные форматы данных и т.д.

Некоторые приложения на этапе разработки не были рассчитаны на работу в составе распределенных ИС, и их интеграция требует внесения изменений в исходный код.

Практически все интеграционные решения предполагают наличие каналов передачи данных устройствами. В отличие от процессов, выполняющихся в пределах одного хоста, в распределенных ИС данные передаются через маршрутизаторы, коммутаторы, общедоступные сети и спутниковые каналы связи, что может приводить к задержкам и рискам потери и искажения данных.

Обычно выделяют четыре базовых механизма интеграции приложений, входящих в состав распределенной ИС [22]:

- разделяемые файлы;
- разделяемая база данных;
- удаленный вызов процедуры и методов;
- обмен сообщениями.

Разделяемые файлы. Несколько приложений имеют доступ к одному и тому же файлу. Одно приложение создает файл, а другое считывает его. Приложения должны согласовать имя файла, его расположение, формат, время записи и считывания, а также процедуру удаления.

Это один из самых старых подходов к интеграции приложений. Основная идея данного подхода состоит в том, что файл рассматривается как универсальный механизм обмена данными. Можно выделить два альтернативных подхода: распределенные файловые системы и системы, основанные на пересылке файлов.

При использовании распределенных файловых систем обмен осуществляется через общие файлы, которые включаются в состав файловых систем взаимодействующих приложений.

Альтернатива рассмотренному выше варианту взаимодействия приложений состоит в том, что файлы, в которых содержится информация, определяющая взаимодействие, пересылаются между хостами. По этому принципу построена, например, система Unix-Unix Copy и ряд других систем. При использовании данного подхода одним из наиболее важных решений является выбор общего формата файлов. В ранних приложениях наиболее распространенным стандартным форматом файлов являлся простой текстовый файл. В современных интеграционных решениях обычно используется XML-формат.

Основное достоинство рассматриваемого подхода — простота, поскольку единственными общедоступными интерфейсами приложений являются создаваемые этими приложениями файлы. В данном случае имеет место слабый вариант связи между приложениями. Кроме того, данный подход обусловливается отсутствием необходимости в привлечении дополнительных средств или пакетов для интеграции. Вместе с тем это приводит к возрастанию нагрузки, ложащейся на плечи разработчиков интеграционного решения. Объединяемые приложения должны использовать общее соглашение об именовании и расположении файлов.

Один из наиболее существенных недостатков передачи файла заключается в сложности синхронизации процессов и сложности разработки кода.

Данный вариант взаимодействия может иметь смысл, если взаимодействие между приложениями носит эпизодический характер.

Разделяемая база данных. Основная идея данного подхода состоит в том, что данные хранятся в центральной базе данных, доступной для всех интегрируемых приложений. Общая база данных обеспечивает согласованность хранящейся в ней информации. Синхронизация доступа к данным реализуется посредством использования, например механизма транзакций.

Самый простой способ реализации общей базы данных состоит в использовании реляционной СУБД с поддержкой SQL. Язык за-

просов SQL поддерживается практически всеми платформами для разработки приложений, что позволяет не беспокоиться о различии в форматах файлов и избавляет от необходимости изучения новых языков программирования и новых технологий. Все вопросы, связанные с интерпретацией данных, могут быть решены на этапе проектирования и реализации интеграционного решения.

С возрастанием числа обращений к общей базе данных она становится узким местом, что приводит к задержкам в работе приложений. Если обращения к разделяемой базе данных осуществляются через локальную и особенно глобальную сеть, то это лишь усугубляет ситуацию с задержками.

Подход к интеграции, основанный на использовании разделяемой базы данных, предполагает использование общей логической структуры данных.

Разделяемая база данных представляет собой неинкапсулированную структуру. Изменения в приложении могут потребовать изменений в структуре базы данных, что, в свою очередь, скажется на других приложениях. Поэтому организации, использующие общую базу данных, обычно без энтузиазма относятся к необходимости ее изменения, что затруднит внедрение новых бизнес-решений.

Удаленный вызов процедуры и методов. С точки зрения интеграции приложений, удаленный вызов процедуры представляет собой применение принципов инкапсуляции данных. Если приложение хочет получить доступ к данным, которые поддерживаются другим приложением, то оно обращается к требуемым данным посредством вызова функции, т.е. каждое приложение самостоятельно обеспечивает целостность своих данных и может изменять их формат, не затрагивая при этом другие приложения.

Удаленный вызов процедуры и работа с удаленными объектами поддерживается множеством технологий, таких как RPC, Java RMI, CORBA, DCOM, .NET Remoting. Несмотря на внешнюю схожесть, удаленный вызов процедуры и вызов локальной функции имеют принципиальные различия, способные оказать существенное влияние на интеграционное решение [18]. Следует отметить, что удаленный вызов процедуры характеризуется самой сильной степенью связывания приложений.

Обмен сообщениями. Идея обмена сообщениями состоит в том, что реализуется асинхронный механизм взаимодействия между приложениями, который позволяет им регулярно обмениваться небольшими порциями информации. Асинхронный обмен сообщениями устраняет большинство недостатков, присущих системам, основанным на вызове удаленных процедур, поскольку для передачи сообщения не требуется одновременной доступности отправителя и получателя. Кроме того, сам факт использования асинхронного обмена данными побуждает разработчиков к созданию приложений, не требующих частого удаленного взаимодействия.

Достоинства, недостатки и условия целесообразности применения четырех перечисленных выше подходов к интеграции приложений изложены в табл. 6.1.

Рассмотрим типы интеграционных задач. В широком смысле под термином «интеграция» можно понимать объединение ИТ-систем и отдельных приложений, входящих в их состав, интеграцию компаний (бизнеса) или людей. В более узком смысле под интеграцией можно понимать объединение отдельных приложений в ИТ-систему, объединение отдельных ИТ-систем в более крупную систему и организацию взаимодействия между отдельными ИТ-системами по принципу B2B.

Интеграция приложений может принимать различные формы, прежде всего можно выделить внутреннюю и внешнюю интеграцию. Внутреннюю интеграцию обычно называют интеграцией корпоративных приложений (Enterprise Application Integration, EAI), а внешнюю — интеграцией бизнес-бизнес (Business-to-Business Application Integration, B2B).

Исходя из последнего определения выделим четыре типовых подхода к решению задачи интеграции [41]:

- интеграция на уровне данных;
- бизнес-функции и бизнес-объекты;
- бизнес-процессы;
- порталы.

Конечно же приведенный выше список ни в коем случае не претендует на звание исчерпывающей классификации задач интеграции. Тем не менее он дает определенное представление о проектах в области интеграционных решений. Некоторые интеграционные задачи объединяют в себе сразу несколько типов интеграции.

Интеграция на уровне данных. Данный подход называют также интеграцией, ориентированной на информацию (Information-Oriented Integration) [41].

Этот подход ориентирован, в первую очередь, на интеграцию данных, которые хранятся в базах данных и обычно имеет целью создание API, позволяющего программисту унифицированным образом работать с множеством БД, которые могут быть территориально разнесены и принадлежать разным производителям. В рамках данного подхода можно выделить, по крайней мере, три группы технологий:

- системы репликации баз данных;
- федеративные базы данных;
- использование API для доступа к стандартным ERP-системам.

В процессе функционирования многих ИТ-систем приложениям часто требуется доступ к одним и тем же данным.

Например, такая информация, как адрес проживания клиента, может использоваться как системой гарантийного обслуживания кли-

Таблица 6.1. **Свойства механизмов интеграции**

Механизма интеграции	Достоинства	Недостатки	Условия целесообразности применения
Разделяемые файлы	Простота. Не требуется использования специальных технологий	Сложности синхронизации процессов и разработки кода. Низкая скорость обмена данными	Приложения разработаны на различных платформах и с использованием различных языков программирования. Обмен данными носит эпизодический характер
Разделяемая база данных	Простота программирования	Сложность разработки структуры общей базы данных. Сложности с масштабируемостью системы	Интегрируемые приложения разработаны на разных платформах и с использованием разных языков программирования, при этом существуют жесткие требования к актуальности и целостности данных
Удаленный вызов процедуры и методов	Высокая скорость обмена. Инкапсуляция данных	Сложность организации асинхронных взаимодействий	Требуется реализовать распределенную функциональность. Жесткие требования по масштабируемости и скорости взаимодействия
Системы, ориентированные на работу с сообщениями	Поддержка асинхронных взаимодействий. Возможность создавать гибкие приложения	Относительно невысокая скорость обмена данными	Требуется реализовать асинхронные взаимодействия. Высокие требования к модифицируемости приложений. Средние требования к скорости обмена данными

ентов, так и подразделениями, занимающимися рекламой. Некоторые из этих систем могут иметь собственные хранилища данных.

При изменении адреса клиента каждая из подсистем должна получить копию обновленной информации. Этого можно добиться с помощью такого типа интеграции, как репликация данных. Существует множество различных способов реализации репликации данных.

Функция поддержки репликаций может быть встроена в СУБД; нужные сведения можно экспортировать в файл для последующего импорта в другой системе, а также переслать внутри сообщений с помощью соответствующего промежуточного ПО. Более подробную информацию об общих принципах реализации механизмов репликации можно найти, например в [17].

Федеративные базы данных (Federated Database System) — это системы, которые позволяют прозрачным для пользователя образом интегрировать множество автономных баз данных, которые могут располагаться на разных хостах сети. Федеративные базы данных называют также виртуальными БД.

Федеративная (виртуальная) БД предоставляет пользователю единый хорошо определенный интерфейс для доступа к распределенным данным, при этом сами данные не перемещаются и не изменяются, т.е. нет препятствий для того, чтобы одна и та же автономная БД входила в состав более чем одной виртуальной БД.

Использование API для доступа к стандартным ERP-системам предполагает использование хорошо определенных интерфейсов для организации взаимодействия создаваемых пользовательских приложений с такими пакетными приложениями, как Enterprise Resource Planning (ERP) системы, SAP, Oracle, PeopleSoft. Обычно это делается посредством использования адаптеров (коннекторов).

Бизнес-функции и бизнес-объекты. Во многих ИТ-системах можно выделить функциональность, которая является общей для нескольких приложений, входящих в состав ИТ-системы. Например, в рассмотренном выше бизнес-приложении эта информация об адресах покупателей.

Каждую из таких функций можно вынести за пределы приложений и реализовать в виде функций совместного использования, доступных всем системам в виде сервисов (служб). В частности, для рассматриваемого примера можно создать, например сервис GetCustomerAddress.

Если организация разрабатывает несколько проектов, в которых используется данная бизнес-функция, то будет разумно сделать ее общей для нескольких проектов.

Отдельные бизнес-функции можно объединить для создания более сложной бизнес-функции, например такой, как постановка изделия на гарантийное обслуживание. Если используется СОА, то данный подход применяется для создания бизнес-сервисов. Если использу-

ется компонентный подход, то создаются бизнес-объекты (бизнес-компоненты).

Совместно используемая бизнес-функция и репликация данных могут преследовать схожие цели. Например, копирование адреса проживания клиента во все требуемые системы можно заменить созданием совместно используемой бизнес-функции GetCustomerAddress. Выбор между двумя разными типами интеграции основывается на многочисленных критериях, таких как степень контроля над интегрируемыми системами (в отличие от помещения информации в базу данных, вызов совместно используемой функции предполагает более глубокое вмешательство в систему) и частота изменения данных (доступ к адресу проживания клиента осуществляется часто, а вот вероятность изменения последнего невысока).

Бизнес-процессы. Данный подход имеет много общего с описанным выше подходом, основанным на использовании бизнес-функций. Основное различие заключается в том, что появляется новый уровень интеграции — уровень бизнес-процессов.

Бизнес-процессы работают поверх уровня сервисов и используют собственный язык для описания последовательности вызова сервисов. Этот язык представляет собой интерпретируемый язык, во многом схожий с такими языками, как Basic или shell.

Бизнес-процессы, обеспечивающие внутреннюю интеграцию, и бизнес-процессы, обеспечивающие B2B-интеграцию, во многом различны. В бизнес-процессах, ориентированных на внутреннюю интеграцию, обычно задействовано достаточно большое количество сервисов. Типовая задача B2B-интеграции состоит в организации взаимодействия между двумя сервисами. Это может быть, например бизнес-процесс приобретения некоторого товара, при котором стороны договариваются о цене и оформляют покупку.

Порталы. Основная функция информационных порталов заключается в обеспечении представления информации из нескольких источников. В качестве таких источников могут выступать, в частности, приложения, которые участвуют в реализации некоторой функции, реализованной средствами бизнес-процессов. Порталы, как правило, реализуют персонифицированный доступ к информации, и его вид может настраиваться пользователем.

Порталы можно рассматривать как графический интерфейс бизнес-процессов, в которых участвуют конкретные пользователи.

6.3. Системы, ориентированные на работу с сообщениями

Системы, ориентированные на работу с сообщениями, реализуют асинхронные механизмы связи и могут поддерживать сохранные механизмы связи.

Кроме рассмотренных ранее систем сообщений, наиболее известными системами, ориентированными на работу с сообщениями, являются следующие:

- MPI;
- системы электронной почты;
- очереди сообщений.

MPI. На работе с сообщениями основана распространенная технология программирования для параллельных вычислительных систем, состоящих из большого числа процессоров, не имеющих общей памяти. В качестве основного способа взаимодействия параллельных процессов, работающих на разных процессорах, используется обмен сообщениями, что и дало название самой технологии — интерфейс передачи сообщений (Message Passing Interface, MPI).

MPI предполагает, что связь происходит в пределах известной группы процессов. Каждый процесс получает идентификатор (processID). Процессы объединяются в группы. Каждая группа имеет уникальный идентификатор (groupID). Идентификатор процесса уникален внутри группы.

Пара идентификаторов processID и groupID однозначно определяет процесс. Каждый из идентификаторов может выступать как в качестве источника, так и в качестве приемника сообщения и используется вместо адреса транспортного уровня. В вычислениях может участвовать произвольное число процессов и групп процессов, которые выполняются одновременно на разных хостах.

MPI представляет собой библиотеку, которая содержит, в частности, примитивы для передачи сообщений, примитивы для приема сообщений и примитивы синхронизации процессов.

MPI-программа — это множество параллельных взаимодействующих процессов. Все процессы создаются только один раз.

В ранних версиях в ходе выполнения MPI-программы создание дополнительных процессов или уничтожение существующих не допускалось. (Позже в версии MPI-2.0 такая возможность появилась.) Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений.

В состав библиотеки входит более ста функций, основные из которых следующие:

- MPI_bsend помещает исходящее сообщение в локальный буфер отправки сообщений;
- MPI_send отсылает сообщение и ждет, пока оно не будет скопировано в локальный или удаленный буфер отправки сообщений;
- MPI_ssend посылает сообщение и ожидает начала его передачи на обработку;
- MPI_sendrecv отсылает сообщение и ожидает ответа;
- MPI_isend передает ссылку на исходящее сообщение и продолжает работу;

- MPI_issend передает ссылку на исходящее сообщение и ожидает начало его передачи на обработку;
- MPI_recv принять сообщение, блокировать работу в случае его отсутствия;
- MPI_irecv проверить наличие входящих сообщений, не блокируя работу.

Данная библиотека поддерживает практически все возможные типы взаимодействий, включая синхронные и асинхронные взаимодействия.

MPI поддерживает работу с языками С и Фортран, однако это совершенно не является принципиальным.

Стандарт MPI фиксирует интерфейсы, которые должны соблюдать как создатели платформ, так и прикладные программисты.

Текущей является версия MPI-2.0, которая появилась в конце 1990-х гг., продолжает использоваться более ранний стандарт MPI-1.1.

Самую подробную информацию о MPI можно получить на сайте [48].

Системы электронной почты. Одним из типовых применений систем работы с сообщениями является система электронной почты. В системе электронной почты хосты работают как пользовательские агенты, которые могут создавать, посылать, принимать и читать сообщения. Каждый хост соединяется с почтовым сервером, который выполняет функции коммуникационного сервера.

Когда пользовательский агент представляет сообщение для передачи на хост, то последний пересылает сообщение на локальный почтовый сервер. Почтовый сервер удаляет сообщение из своего выходного буфера и с помощью сервера имен (DNS) определяет, куда нужно доставить сообщение.

Затем почтовый сервер устанавливает соединение и передает сообщение на целевой почтовый сервер, который, в свою очередь, сохраняет сообщение во входящем буфере получателя, т.е. помещает его в почтовый ящик получателя сообщения. Если искомый почтовый ящик временно недоступен, то сообщение сохраняется на почтовом сервере.

Интерфейс принимающего хоста предоставляет пользовательским агентам сервисы, при помощи которых они могут регулярно проверять наличие сообщений, т.е. пришедшую почту. Агент пользователя может работать либо напрямую с почтовым ящиком пользователя на локальном почтовом сервере, либо копировать сообщения в локальный буфер своего хоста.

Систему электронной почты можно рассматривать как пример сохранной связи.

Очереди сообщений. Использование очередей сообщений позволяет решить многие из проблем, связанных с надежностью и доступностью ИС, в частности, построение систем, работающих по принципу 24/7, т.е. систем, доступных 24 ч в сутки и 7 дней в не-

делю. Подобный эффект достигается за счет того, что приложения в составе ИС обмениваются сообщениями, которые обрабатываются и сохраняются на отдельных работающих круглосуточно серверах.

Такие системы называют системами очередей сообщений (Message-queuing systems, MQ).

Если один компонент ИС хочет послать сообщение другому компоненту, то он посылает данное сообщение MQ, а уж MQ пересылает его адресату.

Основная идея, лежащая в основе MQ, состоит в том, что приложения общаются между собой путем помещения сообщений в очереди. В общем случае эти сообщения передаются по цепочке коммуникационных серверов и достигают места назначения, даже если получатель в момент отправки сообщения был неактивен. Каждое приложение может работать с произвольным числом очередей. Очередь может быть прочитана только связанным с ней приложением, при этом несколько приложений могут совместно использовать одну очередь.

Отправитель может гарантировать только попадание сообщения в очередь получателя, но не может гарантировать то, что сообщение будет действительно прочитано получателем. Отправитель и получатель могут функционировать абсолютно независимо друг от друга. Сообщения в принципе могут содержать любые данные. Адресация осуществляется путем назначения очереди имени, которое должно быть уникальным в пределах системы, в которую направляется сообщение.

Функционирования системы очередей сообщений можно описать с помощью трех примитивов:

- put — добавить сообщение в соответствующую очередь;
- get — извлечь из очереди самое старое сообщение, если очередь пуста, то перейти в режим ожидания и оставаться в нем до момента появлении сообщения в очереди;
- poll — проверить наличие сообщений в очереди, если очередь не пуста, то извлечь из очереди самое старое сообщение, если очередь пуста, то продолжить работу.

Существует две основных модели обмена сообщениями: точка-точка (point-to-point) и публикация-подписка (publish-subscribe).

Модель *точка-точка* применяется тогда, когда отправителю требуется отправить сообщения одному получателю. При использовании данной модели адрес получателя определяется отправителем. Эта модель использует push-модель для работы с сообщениями, т.е. модель «проталкивания» сообщений.

Модель *публикация-подписка* предполагает, что в системе кроме отправителей и получателей имеется несколько очередей, которые называются темами (topics). Отправитель может помещать сообщения в любую тему. Каждый получатель может выражать заинтересованность в получении сообщений из произвольных очередей посредством подписки (subscription).

Модель публикация-подписка применима тогда, когда одному или нескольким компонентам-писателям (publishers) необходимо послать сообщение одному или нескольким компонентам-адресатам-подписчикам (subscribers). Данная модель основана на понятии тем (message topic).

Отправители посылают сообщения в темы, и все подписчики данной темы получают эти сообщения.

Обе модели основаны на понятии очередь сообщений (message queue).

Сообщения могут быть помещены только в локальные очереди отправителя, т. е. в очереди, находящиеся на том же самом хосте. Эта очередь называется исходящей очередью (source queue). Аналогично, прочитанные сообщения могут быть прочитаны только из локальных очередей. Таким образом, набор очередей представляет собой распределенную систему, разнесенную на множество хостов.

Выбор первой или второй модели определяется спецификой задачи.

Очереди управляются менеджерами очередей (queue managers). Обычно менеджер очередей взаимодействует непосредственно с отправляющими и принимающими сообщения приложениями. Существуют, однако, и специализированные менеджеры очередей, которые работают как маршрутизаторы, или ретрансляторы: они перенаправляют приходящие сообщения другим менеджерам очередей.

Практически все ведущие производители ИС, такие как Microsoft, IBM, Oracle и др., предлагают собственные системы работы с очередями сообщений [17].

6.4. Язык описания бизнес-процессов BPEL

Общие сведения. Web-сервисы представляют собой интерфейсы для доступа к автономным, модульным приложения. Для того чтобы обратиться к Web-сервису, необходимо послать SOAP-послание по определенному адресу (XML-документ). При этом не имеет значения, каким именно образом формируются эти послания.

BPEL — это язык, который позволяет описывать бизнес-процесс в терминах некоторой последовательности обращения к Web-сервисам. BPEL, по существу, является скриптовым языком программирования, поддерживающим синхронные и асинхронные взаимодействия, параллельное выполнение и обработку исключений. Программа (приложение), написанное на языке BPEL, является XML-документом.

Язык BPEL позволяет задавать бизнес-процессы, при этом приложение, написанное на языке BPEL, можно рассматривать как Web-сервис, и к нему можно обращаться посредством посылки SOAP-посланий.

BPEL является интерпретируемым языком и для его использования необходимо наличие процессора (движка).

BPEL был разработан на базе двух более ранних языков описания бизнес-процессов WSFL и Xlang фирмы IBM и Microsoft соответственно. Первый стандарт BPEL (BPEL4WS) сразу в версиях 1.0 и 1.1 появился в 2003 г. На момент написания данной книги последней была версия WS-BPEL 2.0, которая появилась в апреле 2007 г. и доступна на сайте http://docs.oasis-open.org/wsbpel/2.0/. В июне 2007 г. были опубликованы спецификации BPEL4People и WS-HumanTask, где описывались возможности реализации в BPEL взаимодействия с людьми. Последние версии этих документов можно найти по адресу http://docs.oasis-open.org/.

Основу BPEL составляют три ключевых свойства: асинхронность, координация потоков и управление исключительными ситуациями.

Асинхронность имеет дело с асинхронными взаимодействиями, корреляцией сообщений и надежностью. Поддержка асинхронности необходима для разрешения Web-сервисов в сценариях интеграции и является обязательной для оптимального использования рабочего времени (для лучшего распределения обработки она позволяет пользователям вмешиваться в течение бизнес-потока или задержанной пакетной обработки). За счет разделения запросов на обслуживание и соответствующих им откликов асинхронность повышает масштабируемость и помогает избежать узких мест при выполнении приложения. Кроме того, она допускает непрерываемое выполнение, когда сервисы временно недоступны и когда клиенты работают в автономном режиме или отключены.

Координация потоков включает в себя параллельные потоки выполнения, образцы соединений и динамические потоки. В реальных приложениях бизнес-потоки могут включать образцы сложных взаимодействий как с синхронными, так и с асинхронными сервисами. Координация потока включает интерфейс с WSDL, действия потока, переменные XML и отвечает за координацию. BPEL использует WSDL для обращения к сообщениям, вызванным операциям и типам портов. Действия с потоком используют общие переменные XML, так что компенсационные обработчики (compensation handlers) должны сохранять снимки данных, которые могут быть использованы обработчиком. Компенсационные обработчики могут отменить шаги, которые были уже завершены.

Управление исключительными ситуациями имеет дело с синхронными ошибками, асинхронным управлением исключительными ситуациями и компенсацией бизнес-транзакций. Для того чтобы автоматизировать бизнес-процессы, большие усилия сосредоточены на управлении исключительными ситуациями, и BPEL упрощает управление исключительными ситуациями для Web-сервисов. При возникновении исключительных ситуаций вызываются локальные обработчики ошибки, связанные с Web-сервисами, и асинхронные сервисы уведомляются об этих исключительных ситуациях.

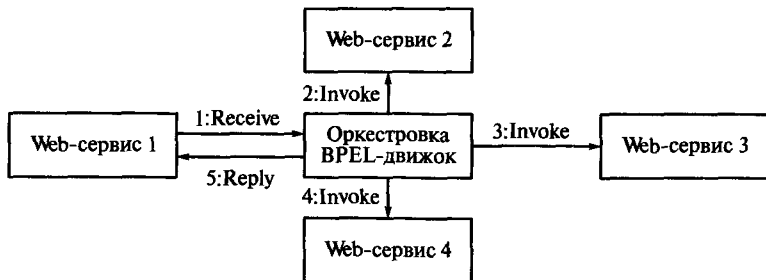


Рис. 6.1. Схема оркестровки

Подходы к объединению Web-сервисов в бизнес-процессы. Принято выделять два основных подхода к объединению Web-сервисов в бизнес-процессы: оркестровка (Orchestration) и хореография (Choreography).

Идея *оркестровки* состоит в том, что в системе имеется единственный BPEL-процессор (движок), который выполняет функции интерпретатора BPEL-файлов. Для внешнего мира BPEL-процессор доступен как Web-сервис. Получив запрос, движок уже от своего имени рассылает SOAP-послания Web-сервисам, участие которых необходимо для реализации бизнес-процесса. Задействованные веб-сервисы не знают, что они вовлечены в бизнес-процесс более высокого уровня. Только движок обладает полной информацией о выполняемой задаче, и поэтому оркестровка является централизованным механизмом с явным определением операций и порядком инициирования работы Web-сервисов (рис. 6.1).

Использование *хореографии* (рис. 6.2), напротив, не предполагает использование центрального координатора. Каждому из Web-сервисов, участвуюших в хореографии, известно, когда следует выполнить те или иные операции и с какими Web-сервисами необходимо взаимодействовать.

Хореография представляет собой совместное действие, ориентированное на обмен сообщениями при реализации бизнес-процессов, в которых участвуют несколько организаций. При этом все

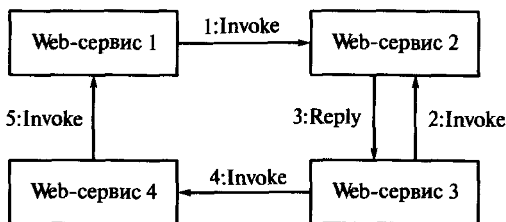


Рис. 6.2. Схема хореографии

участники должны знать бизнес-процесс, выполняемые операции, сообщения, которыми они обмениваются, и синхронизировать эти обмены сообщениями.

На первый взгляд, кажется, что оркестровка и хореография представляют собой альтернативные подходы к организации бизнес-процессов, однако, если предположить, что за Web-сервисами, участвующими в оркестровке, стоят отдельные движки, то различия между оркестровкой и хореографией уже не столь очевидны.

Обычно оркестровка используется для создания исполняемых BPEL-файлов, а хореография — как средство для описания протоколов, описывающих взаимодействие, например между организациями. При этом не предполагается использовать эти описания в качестве исполняемых файлов.

BPEL поддерживает два различных способа описания бизнес-процессов, которые поддерживают оркестровку и хореографию:

- исполняемые процессы (Executable processes) позволяют определять точную детализацию бизнес-процессов. Исполняемый процесс моделирует поведение участников определенного бизнес-взаимодействия, в сущности, моделируя частный поток работ. Исполняемые процессы находятся в парадигме оркестровки и могут быть выполнены механизмом оркестровки;
- абстрактные бизнес-протоколы (Abstract business protocols) определяют обмен публичными сообщениями между участниками. Они не включают внутренние детали потока процессов, не являются выполнимыми и находятся в парадигме хореографии.

Алгоритмически полный язык BPEL, система типов которого соответствует языку XML, обладает выразительными управляющими конструкциями, поддержкой параллельного исполнения, обработкой исключений, поддержкой транзакций, взаимодействия процессов между собой и другими возможностями.

BPEL предоставляет такие механизмы управления вычислительным процессом, как присваивание значений переменным, условные операторы, возможность организации циклов, работа с событиями и др.

Описание процессов в BPEL. BPEL-процесс состоит из активностей (activities): базовых (basic) и структурированных (structured).

Базовые активности не включают в себя другие активности и выполняют такие элементарные действия, как прием сообщения от партнера или выполнения элементарных действий с данными.

Важнейшие базовые активности ориентированы на получение и отправку сообщений. Это активности invoke, receive и reply, которые определяют два способа взаимодействия: асинхронное и синхронное.

Активность типа *invoke* предполагает одностороннее взаимодействие. Вызывающая сторона посылает сообщение и продолжает функционирование. Получение ответа не предусматривается.

Синхронное взаимодействие реализуется с помощью пары активностей. Сервис реализует активность типа *receive* — находится в состоянии ожидания запроса. Получив запрос, сервер формирует ответ посредством реализации активности *reply*. До получения ответа вызывающая сторона находится в состоянии ожидания, т.е. в заблокированном состоянии.

Имеются и другие базовые активности — ориентированные на присвоение значений переменным (assign), остановка реализации сервиса (terminate), отсутствие действий (empty), активность типа pick и Event Handler, ориентированные на поддержку работы с событиями.

Структурированные активности включают в себя другие активности и обеспечивают реализацию бизнес-логики.

В рамках BPEL определен достаточно широкий набор структурированных активностей, основными из которых являются следующие:

- задание последовательности выполнения действий (<sequence>);
- цикл (<while>);
- выбор одного из нескольких альтернативных маршрутов (<pick>);
- параллельное выполнение (<flow>);
- обработка ошибок <throw> и <catch>;
- объединение нескольких действий (<scope>) и др.

При автоматизации бизнес-процессов значительное внимание уделяется управлению исключительными ситуациями, BPEL упрощает управление исключительными ситуациями для Web-сервисов. При возникновении исключительных ситуаций вызываются локальные обработчики ошибок, связанные с Web-сервисами, и асинхронные сервисы получают соответствующие уведомления. Существуют специальные компенсационные обработчики (compensation handlers), сохраняющие снимки данных, которые могут быть использованы обработчиком. Компенсационные обработчики могут отменить шаги, которые были уже завершены. Кроме обработки ошибок и тайм-аутов оркестрованные Web-сервисы должны гарантировать доступность ресурсов при выполнении длительных распределенных транзакций. Традиционные транзакции обычно не вполне пригодны для реализации длительных распределенных бизнес-операций, поскольку не позволяют блокировать необходимые ресурсы на продолжительный срок. При использовании компенсирующих транзакций каждый метод включает в себя операцию отмены, которую координатор транзакций может вызвать в случае необходимости (под компенсирующей транзакцией понимают возможность отката операции при ее отмене процессом или потребителем).

Структура BPEL-документа. В самом общем виде структура BPEL документа выглядит следующим образом:

```
<process name="MyBusinessProcess" … >
     <partnerLinks>
```

```
    <! -- Определение партнерских связей -->
    </partnerLinks>
    <variables>
    <!-- Определение переменных -->
    </variables>
    <sequence>
    <!-- Определение основной части BPEL
      бизнес-процесса -->
    </sequence>
</process>
```

Контейнер, заключенный в теги <process>, включает следующие вложенные элементы:

- партнерские связи <partnerLinks>;
- описания переменных <variables>;
- описание последовательности обращений к Web-сервисам <sequence>.

В качестве примера рассмотрим простейший бизнес-процесс. Допустим, что сотруднику организации требуется оформить приобретение некоторого оборудования (картриджа для принтера, компьютера или автомата для приготовления кофе). Для того чтобы приобрести оборудование, сотрудник должен получить разрешение соответствующего менеджера. Затем просматривается возможность приобретения данного оборудования у разных поставщиков, в качестве которых выступают Компания А и Компания В. Информация о лучшем с точки зрения цены предложении возвращается сотруднику.

Структура данного бизнес процесса показана на рис. 6.3. Бизнес-процесс представляет собой сервис, реализующий асинхронный режим функционирования. Менеджер представлен синхронным сервисом, работающим по принципу запрос-ответ. Компания А и Компания В представлены асинхронными сервисами.

Функционирование сервиса выглядит следующим образом. Когда сотруднику (клиент) потребуется заказать оборудование, он запускает клиентское приложение, которое может быть, например портлетом, вводит информацию о необходимом оборудовании и отсылает запрос. Клиентское приложение формирует асинхронный запрос к основному сервису, который назовем Сервисом закупки (PurchaseService). После этого формируется синхронный запрос к сервису Менеджер (ManagerService), который возвращает послание с разрешением на покупку оборудования. В качестве Менеджера может выступать либо приложение, которое может, например, обратиться к базе данных с проверкой наличия средств на счете подразделения, либо реальный человек. В последнем случае у менеджера появляется информация о том, что на его имя пришел запрос. В последнем случае Менеджер открывает соответствующий портлет и заполняет требуемые поля. Если Менеджер не дает разрешения, то вызывается активность типа terminate.

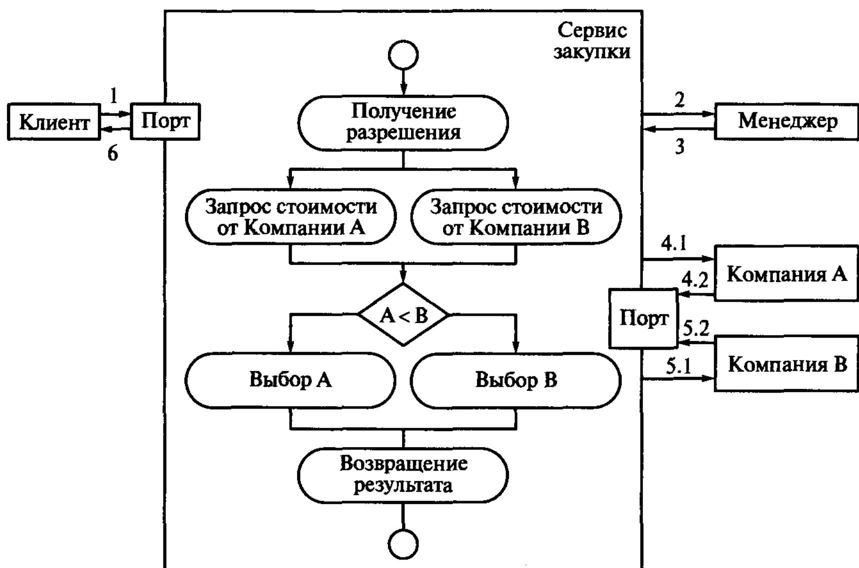


Рис. 6.3. Пример структуры бизнес-процесса

После получения разрешения формируются запросы к серверам компаний, занимающимися продажей требуемого оборудования. В рассматриваемом примере формируются два параллельных асинхронных запроса к сервисам компаний. После того как обе компании прислали свои цены на требуемое оборудование, выбирается минимальная цена и сообщается пользователю.

Рассмотрим более подробно проектирование данного сервиса. Далее будет рассмотрен процесс «ручного» проектирования BPEL-процесса, хотя на практике для этого используются инструментальные средства разработки.

Разработка BPEL-процесса обычно включает в себя следующие этапы:

- определение Web-сервисов, участвующих в разрабатываемом BPEL-процессе;
- разработка WSDL-описания BPEL-процесса;
- определение партнерских связей;
- декларирование переменных;
- разработки логики процесса.

В рассматриваемом примере бизнес-процесс работает с тремя сервисами: Менеджером (ManagerService), Компанией А (CompanyAService), Компанией В (CompanyBService).

Web-сервис Менеджер (ManagerService) имеет порт типа EquipmentPurchcePT, через который при помощи операции GetPermission может быть получено разрешение на покупку. Операция возвращает

разрешение или отказ в виде строки символов. Данный сервис является синхронным. Запрос помещается в сообщение PermissionRequestMessege, а ответ в сообщение типа PermissionResponseMessege.

Web-сервис Компания А (CompanyAService) является асинхронным; он определяет два типа портов: первый — это PriceListAPT, который использует операцию GetPrice для проверки наличия оборудования и цены. Для возврата результата Web-сервис определяет отдельный порт типа — CompanyACallbackPT. Этот тип порта определяет операцию CompanyACallback. Для пересылки результата используется сообщение типа PriceResponseMessage.

Web-сервис Компания В (CompanyBService) также является асинхронным; он определяет два типа портов: первый — это PriceListBPT, который использует операцию GetPrice для проверки наличия оборудования и цены. Для возврата результата Web-сервис определяет отдельный порт типа CompanyBCallbackPT. Этот тип порта определяет операцию CompanyBCallback. Для пересылки результата используется сообщение типа PriceResponseMessage.

Хотя Web-сервисы Компания А и Компания В определяет по два типа портов, они реализуют только по одному порту (PriceListAPT и PriceListBPT), а порты CompanyACallbackPT и CompanyBCallbackPT реализуется BPEL-процессом — клиентом этого Web-сервиса.

Затем необходимо представить сам процесс приобретения оборудования как Web-сервис, что можно сделать посредством составления его WSDL-описания.

Процесс имеет порт типа PurchaseEquipmentPT. Для обращения к порту используется операция BuyEquipment и сообщение формата BuyEquipmentRequestMessage.

Кроме того, на стороне клиента должен иметься порт ClientCallbackPT, в который отправляется клиенту ответ, с результатами обработки запроса. Для этого используется операция SendPurchaseInfo и сообщение типа BuyEquipmentResponseMessage.

Следующий шаг — определение партнерских связей, которые устанавливают взаимодействия между BPEL-процессом и используемыми Web-сервисами. В рассматриваемом примере определим следующие типы партнерских связей:

• purchaseLT используется для описания асинхронного взаимодействия между клиентом и собственно BPEL-процессом, определяется WSDL-описанием BPEL-процесса;

• managerLT используется для описания синхронного взаимодействия между BPEL процессом и Web-сервисом ManagerService;

• companyALT описывает асинхронное взаимодействие между BPEL-процессом и Web-сервисом Компания А. Этот тип партнерской связи определен в WSDL Web-сервиса CompanyAService;

• companyBLT описывает асинхронное взаимодействие между BPEL-процессом и Web-сервисом Компания В. Этот тип партнерской связи определен в WSDL Web-сервиса CompanyBService.

За партнерской связью закрепляются одна или две роли. Для синхронных операций определяется одна роль, а для асинхронной определяются две роли.

Для асинхронных операций первая роль описывает вызов операции клиентом, а вторая роль — процесс возвращения ответа на запрос, который обычно называют обратным вызовом (callback).

Типы партнерских связей определяются в WSDL-описании в специальном пространстве имен (namespace).

Для связи purchaseLT имеется две роли. Первая роль описывает сервис, к которому обращается служащий. Клиент использует данный тип порта для того, чтобы связаться с BPEL-сервисом. Вторая роль описывает клиента, к которому обращается BPEL-процесс при реализации обратного вызова. Описание рассмотренных ролей выглядит следующим образом:

```
<plnk:partnerLinkType name="purchaseLT">
<plnk:role name="purchaseService">
  <plnk:portType name="tns: PurchaseEquipmentPT" />
</plnk:role>
<plnk:role name="purchaseServiceCustomer">
  <plnk:portType name="tns:ClientCallbackPT" />
</plnk:role>
</plnk:partnerLinkType>
```

Второй тип связи managerLT используется для описания синхронной связи между BPEL-процессом и Web-сервисом Менеджер и определяется в WSDL-описании Web-сервиса менеджера. Поскольку взаимодействие синхронно, то имеется только одна роль:

```
<plnk:partnerLinkType name="managerLT">
<plnk:role name="purchasePermissionService">
  <plnk:portType name="tns: EquipmentPurchacePT" />
</plnk:role>
</plnk:partnerLinkType>
```

Оставшиеся типы партнерских связей, companyALT и companyBLT, используются для описания асинхронных связей между BPEL-процессом и Web-сервисами Компания А и Компания В. Для Компании А описание имеет вид:

```
<plnk:partnerLinkType name="companyALT">
<plnk:role name="purchaseAService">
  <plnk:portType name="tns: PriceListAPT" />
</plnk:role>
<plnk:role name="equipmentCustomer">
  <plnk:portType name="tns:CompanyACallbackPT" />
</plnk:role>
</plnk:partnerLinkType>
```

Для Компании B описание аналогично.

Создание бизнес-процесса. BPEL-процесс работает следующим образом. BPEL-процесс запускается при поступлении входного сообщения от клиента. После этого происходит вызов сервиса Менеджер. Это синхронный вызов, поэтому выполнение процесса приостанавливается до получения ответа. После того как ответ получен, посылаются запросы к Web-сервисам Компании A и Компании B. Это асинхронные вызовы, выполняемые параллельно. После того как ответы с информацией о стоимости требуемого оборудования и полученные предложения сравниваются по цене в рамках бизнес процесса, выбирается лучшее предложение. Затем информация о выборе передается служащему.

Рассмотренный выше алгоритм описывается в блоке, заключенном в теги <sequence> </sequence>, в котором содержится описание последовательности действий.

Однако перед тем как описывать последовательность действий, требуется определить переменные.

Переменные (Variables) в BPEL-процессах используются для хранения и повторного использования значений, полученных в от сервисов.

Для работы с сообщениями используется XPath.

В рассматриваемом примере используются следующие переменные:

- EquipmentRequest;
- PermissionRequest;
- PermissionResponse;
- EquipmentDetails;
- AResponse;
- BResponse;
- Response2employer.

Для каждой переменной требуется определить ее тип. При определении переменных можно использовать типы, используемые в WSDL-описаниях, а также простые типы XML Schema. В рассматриваемом примере использованы для всех переменных типы WSDL-сообщений:

```
<variables>
  <!—входные данные для бизнес процесса -->
  <variable name="EquipmentRequest"
    messageType="prch: BuyEquipmentRequestMessage"/>
  <!—входные данные для запроса о разрешении
    приобретения оборудования -->
  <variable name="PermissionRequest"
    messageType="prch: PermissionRequestMessage"/>
  <!—выходные данные от менеджера -->
  <variable name="PermissionResponse"
```

```
    messageType="prch: PermissionResponseMessage"/>

  <!—входные данные для компании A -->
  <variable name=" EquipmentDetails"
    messageType="cmp: PriceRequestMessage"/>
  <!—выходные данные от компании A -->
  <variable name=" AResponse"
    messageType=" cmp: PriceResponseMessege"/>

  <!—входные данные для компании B -->
  <variable name=" EquipmentDetails"
    messageType="cmp: PriceRequestMessage"/>
  <!—выходные данные от компании B -->
  <variable name=" BResponse"
    messageType=" cmp: PriceResponseMessege"/>
  <!-- выходные данные от BPEL процесса -->
  <variable name="Response2employer"
    messageType="cmp: BuyEquipmentResponseMessage"/>
</variables>
```

Основная часть BPEL-процесса заключена в теги <sequence> и описывает порядок, в котором вызываются партнерские Web-сервисы. Запуск BPEL-процесса происходит при поступлении в порт PurchaseEquipmentPT сообщения типа BuyEquipmentRequestMessage. Параметры запроса сохраняются в переменной EquipmentRequest:

```
<sequence>
  <!—Получение от клиента запроса на приобретение
    оборудования -->
  <receive partnerLink="purchaseLT"
    portType="prch:PurchaseEquipmentPT"
    operation="BuyEquipment"
    variable="EquipmentRequest"
    createInstance="yes" />
...
```

<receive> ожидает, когда клиент вызовет операцию BuyEquipment и сохраняет параметры запроса в переменной EquipmentRequest. Строка createInstance="yes" указывает на то, что требуется создать новый экземпляр бизнес-процесса.

Далее требуется вызвать сервис Менеджер. Для этого необходимо сформировать для него входное сообщение посредством копирования части сведений о требуемом оборудовании:

```
...
  <!—Подготовка входного сообщения для менеджера -->
  <assign>
  <copy>
```

```
    <from variable="EquipmentRequest"
      part="EquipmentType"/>
    <to variable="PermissionRequest"
      part="EquipmentType"/>
  </copy>
  </assign>
...
```

Теперь может быть вызван сервис ManagerService. С этой целью используется <invoke>. Применяется партнерская связь *managerLT* и вызывается операция GetPermission; осуществляется обращение к порту типу EquipmentPerchasePermissionPT. Поскольку это синхронное обращение, то запрос ждет ответа, который сохраняется в переменной PermissionResponse:

```
...
  <!— синхронное обращение к менеджеру
      для получения разрешения -->
    <invoke partnerLink="managerLT"
      portType="prch:EquipmentPurchasePermissionPT"
      operation="GetPermission"
      inputVariable="PermissionRequest"
      outputVariable="PermissionResponse" />
...
```

Следующий шаг должен вызвать Web-сервисы обеих компаний. Для этого требуется подготовить входные сообщения (для простоты будем считать, что они идентичны для обоих сервисов). Сообщение PermissionRequest может состоять из нескольких частей, но нас будет интересовать только одна часть — тип оборудования, полученная из запроса клиента:

```
...
  <!—Подготовка запроса к сервисам компаний -->
  <assign>
  <copy>
    <from variable="PermissionResponse"
      part="EquipmentType"/>
    <to variable=" EquipmentDetails"
      part="EquipmentType"/>
  </copy>
  </assign>
...
```

Входные данные включают в себя сведения, которые требуются Web-сервисам компаний. Поскольку они заданы в одном и том же формате, их можно передать непосредственно (используя простую копию). В реальной же ситуации обычно необходимо выполнить пре-

образование. Это можно было бы сделать, использовав либо XPath с выражением <assign>, либо сервис преобразования (типа механизма XSLT), либо возможность преобразования, обеспеченную определенными BPEL-серверами.

Теперь можно вызывать Web-сервисы компаний поставщиков. Для параллельного вызова сервисов применяется активность <flow>. Оба сервиса асинхронные и поэтому реализуются активность <invoke> и активность <receive>.

Результаты, полученные от компаний, сохраняются в переменных AResponse и BResponse соответственно:

```
...
   <!—Параллельное обращение к сервисам компаний -->
   <flow>

   <sequence>
   <!-- Асинхронный вызов Web-сервиса
        Компании A и ожидание ответа -->

   <invoke partnerLink="CompanyALT"
     portType="spl:PriceListAPT"
     operation="GetPrice"
     inputVariable="EquipmentDetails" />

   <receive partnerLink="CompanyALT"
     portType="spl:FlightCallbackPT"
     operation="CompanyACallback"
     variable="AResponse" />
   </sequence>

   <sequence>
   <!-- Асинхронный вызов Web-сервиса
        Компании B и ожидание ответа -->
   <invoke partnerLink=" CompanyBLT"
     portType="spl:PriceListAPT"
     operation="GetPrice"
     inputVariable="EquipmentDetails" />

   <receive partnerLink=" CompanyBLT"
     portType="spl:FlightCallbackPT"
     operation=" CompanyBCallback"
     variable="BResponse" />
   </sequence>

</flow>
...
```

На следующем шаге осуществляется выбор лучшего по стоимости предложения посредством использования активности <switch>:

```
...
   <!-- Выбор лучшего варианта и формирование
      ответа клиенту -->
   <switch>

   <case condition="bpws:getVariableData('AResponse',
        'confirmationData','/confirmationData/Price')
      <= bpws:getVariableData(' BResponse',
        'confirmationData','/confirmationData/Price')">

   <!-- Вариант от компании A -->
   <assign>
   <copy>

      <from variable="AResponse" />
      <to variable="Response2employer" />
      </copy>
      </assign>
   </case>

   <otherwise>
      <!-- Вариант от компании B -->
      <assign>
      <copy>
         <from variable="BResponse" />
         <to variable="Response2employer" />
      </copy>
      </assign>
   </otherwise>
   </switch>
...
```

В элементе <case> для сравнения предложений используется BPEL-функция getVariableData и определяется имя переменной. Цена находится в разделе confirmationData, который является единственной частью сообщения, но это все же требуется определить. Также нужно определить выражение вопроса, чтобы задать местонахождение элемента цены. Здесь это делается простым выражением XPath 1.0.

Если предложение от A лучше, чем от B, копия переменной AResponse помещается в переменную Response2employer. В противном случае копируется переменная BResponse. Затем посредством использования активности <invoke> результаты бизнес-процесса возвращаются клиенту. Для этого служит партнерская связь purchaseLT и вызывается операция SendPurchaseInfo по типу порта ClientCallbackPT:

```
...
  <!—Формирование ответа клиенту -->
  <invoke partnerLink="purchaseLT"
    portType="emp:ClientCallbackPT"
    operation="SendPurchaseInfo"
  inputVariable="Response2employer" />
</sequence>
</process>
```

Средства разработки. Поскольку BPEL фактически представляет собой диалект языка XML, скрипт BPEL можно создавать либо вручную, либо, что более предпочтительно, воспользоваться одним из существующих программных инструментов для генерации скриптов. Скрипт BPEL — это XML-документ, соответствующий структуре BPEL-документа. Он интерпретируется во время исполнения процессором BPEL, который выявляет ключевые слова и выполняет соответствующую обработку.

Корректная и полная реализация стандарта BPEL должна поддерживать следующий набор стандартов Web-сервисов: WSDL 1.1, XML Schema 1.0, XPath 1.0, WS-Addressing, UDDI v2.0, WS-Security (необязательно).

Среди свободно распространяемых программных продуктов, предоставляющих возможности генерации скриптов BPEL, можно выделить Netbeans BPEL Designer — компонент Netbeans Enterpise Pack, представляющий собой бесплатное дополнение к NetBeans и добавляющий к его функциональным возможностям средства визуального проектирования приложений с сервис-ориентированной архитектурой. NetBeans Enterprise Pack содержит средства для проектирования BPEL, схем XML, WSDL файлов, а так же средства обеспечения безопасности Web-сервисов. BPEL Designer позволяет оркестрировать Web-сервисы, а именно создавать, разрабатывать, запускать и тестировать бизнес-процессы BPEL, соответствующие спецификации WS-BPEL 2.0. Кроме того, данный продукт предоставляет визуальные средства, позволяющие быстро и эффективно оркестрировать Web-сервисы в рамках бизнес-взаимодействия.

Среди платных программных продуктов известен Oracle BPEL Process Manager (BPM) — инфраструктурное решение для проектирования, размещения и управления бизнес-процессами по стандарту BPEL.

6.5. Бизнес-правила

Любая организация использует в процессе функционирования определенный набор законов, постановлений правительства, промышленных стандартов и корпоративных политик, которые в совокупности и называются называются ***бизнес-правилами*** (business

rules). Наблюдение за их выполнением и соблюдением может осуществляться как непосредственно людьми, так и ИТ-системами.

Под ***бизнес-логикой*** обычно понимают совокупность правил, принципов, зависимостей, поведения объектов предметной области системы. Иначе можно сказать, что бизнес-логика применительно к ИТ-системам — это реализация правил и ограничений автоматизируемых операций. Термин «бизнес-логика» часто используют как синоним термина «логика предметной области» (Domain Logic).

К бизнес-логике относятся, например, формулы расчета ежемесячных выплат по ссудам (в финансовой сфере), автоматизированная отсылка электронного письма руководителю проекта по окончанию выполнения частей задания всеми подчиненными (в системах управления проектами) и т. п.

Согласно определению Business Rules Group (1993) «бизнес-правило — это положение, определяющее или ограничивающее какие-либо стороны бизнеса; его назначение состоит в том, чтобы защитить структуру бизнеса, контролировать или влиять на его операции». Целые методологии разработаны специально для создания и документирования бизнес-правил и их применения в ИС. Если не создается система, которая в значительной степени управляется бизнес-правилами, тщательно разработанная методология не нужна. Достаточно выявить и задокументировать относящиеся к системе правила и связать их с конкретными функциональными требованиями.

Бизнес-правила берут начало вне контекста какой-либо конкретной ИТ-системы и являются одним из главных источников функциональных требований к ИТ-системам, поскольку они определяют те возможности, которыми должна обладать система для выполнения правил.

Не все организации рассматривают собственные важнейшие бизнес-правила как ценность, которой они являются на самом деле. Если эта информация не задокументирована и не хранится должным образом, то она существует только в головах сотрудников. Сотрудники и разработчики ИТ-систем могут по-разному понимать правила, в результате чего отдельные программные системы могут по-разному выполнять одно и то же бизнес-правило. Если известно, где и каким образом приложение реализует относящиеся к нему бизнес-правила, то модифицировать приложения в случае изменения соответствующих правил гораздо проще.

В больщих организациях часто бывает очень трудно определить, какое именно приложение ответственно за реализацию того или иного бизнес-правила.

Создание общих правил и наличие централизованного хранилища бизнес-правил упрощает согласованную реализацию приложений, на которые эти правила влияют.

Типы бизнес-правил. Известны разные подходы к классификации правил. Схема простейшей классификации, показанная на рис. 6.4, выделяет пять типов бизнес-правил.

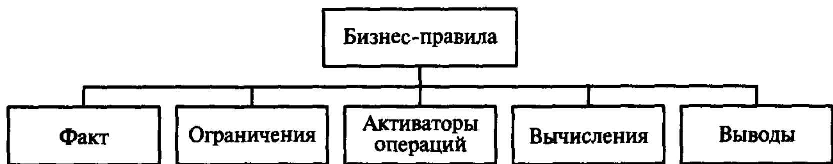


Рис. 6.4. Схема простейшей классификации, типов бизнес-правил

Факты (facts) представляют собой верные утверждения о бизнесе, которые описывают связи и отношения между значимыми бизнес-терминами. Факты также называют инвариантами, т.е. неизменными истинами о сущности данных и их атрибутах. Бизнес-правила могут ссылаться на факты, однако факты обычно не преобразуются напрямую в функциональные требования к системе. Сведения о сущностях, важных для системы, применяют в моделях данных, создаваемых аналитиком или архитектором.

Можно привести следующие примеры фактов:

- на каждом продукте имеется наклейка с уникальным штрих-кодом;
- каждый сотрудник имеет пропуск;
- каждый элемент заказа содержит данные о продукте и сроке поставки;
- стоимость билетов не возвращается, если пассажир опаздал на рейс.

Ограничения (constraints) определяют состав операций, которые может выполнять как отдельные пользователи, так и ИТ-система. При описании ограничивающего бизнес-правила обычно используются следующие глаголы «должен», «не должен», «может», «не может», которые могут встречаться в разных комбинациях;

- постоянный читатель библиотеки может взять для прочтения до пяти книг;
- при входе в зоопарк взрослый может провести бесплатно до двух детей;
- все входы в медицинские учреждения должны соответствовать правительственным постановлениям, касающимся использования их людьми с ограниченными физическими возможностями.

При реализации проектов в области создания ИС обычно имеется большое количество разных ограничений, например, менеджеры проектов должны соблюдать сроки поставок и лимиты по бюджету. Ограничения уровня проекта фиксируются в документации по управлению проектом, а ограничения на проектирование и реализацию, уменьшающие доступные разработчику возможности, обычно записаны в спецификации требований к ИС или в спецификации архитектуры.

Активаторы операций (action enabler) можно определить как правило, при определенных условиях приводящее к выполнению

каких-либо действий (action enabler). Правило может управлять программными функциями либо действия могут выполняться вручную. Условия, определяющие начало выполнения операции, обычно формулируются в терминах булевой алгебры. Для этого бывает полезно использовать таблицы.

В самом общем виде активатор можно описать выражением вида «Если < наступило определенное событие или выполняется некоторое условие >, то <что-то происходит>».

В качестве примеров активаторов могут служить следующие бизнес-правила:

• если на складе имеются запрошенный покупателем товар, то его следует предложить запросившему товар лицу;

• если на складе отсутствует запрашиваемый товар, то покупателю следует предложить оставить заявку;

• если клиент заказал книгу по определенной тематике, то ему следует предложить другие книги по данной тематике, прежде чем принять заказ.

Вычисления (computations) — это бизнес-правила, выполняемые с использованием математических формул и алгоритмов. При этом вычисления могут выполняться по внешним для организации правилам, например по формулам исчисления налогов. Бизнес-правила данного типа могут описываться разными способами: в форме текстового описания, в виде формул или в виде таблиц.

Ниже дано несколько примеров бизнес-правил для вычислений в текстовой форме:

• стоимость доставки товара составляет 10 %, но не менее 300 руб.;

• при покупке от 6 до 10 единиц товара скидка составляет 3 %, при покупке более 10 — 5 %.

Вывод (inference) — это правило, которое позволяет создавать новый факт на основе других фактов или вычислений. Часто вывод записывается в формате «если — то». Следует отметить, что данный формат применяется также в активаторах операций. Разница состоит в том, что в случае вывода раздел «то» описывает не действие, а заключает в себе факт.

Приведем несколько примеров выводов:

• если поставщик не может поставить заказанный товар в течение 30 календарных дней с момента получения заказа, то заказ считается невыполненным;

• если платеж не поступил в течение 5 банковских дней с момента отправки счета, счет считается недействительным.

Документирование бизнес-правил. Значительная часть бизнес-правил влияет на множество приложений, и поэтому в рамках организации целесообразно управлять этими правилами не на уровне отдельных проектов, а на корпоративном уровне. Если число используемых правил невелико, то для начала бывает достаточно простого

каталога бизнес-правил, при увеличении числа бизнес-правил следует создать базу данных таких правил. По мере того как при работе над приложением определяются новые правила, их добавляют в каталог, а не вписывают в документацию конкретного приложения или, что еще хуже, только в его код. По мере приобретения опыта выявления и документирования бизнес-правил можно переходить к использованию структурированных шаблонов для определения правил разных типов, в которых описываются образцы ключевых слов и разделов, позволяющих согласованно структурировать правила. Использование шаблонов упрощает хранение правил в базе данных и коммерческих инструментах для управления бизнес-правилами.

Источниками правил могут быть корпоративные политики, политики менеджмента, профильные специалисты и прочие лица и документы, например правительственные постановления, а также имеющийся код программных продуктов и определения баз данных.

В настоящее время отсутствует единый стандарт на формат бизнес-правил, однако разработан целый ряд сопутствующих стандартов, наиболее известными из которых являются следующие:

- OMG Business Motivation Model (BMM) определяет применение стратегии, процессов и правил в бизнес-моделировании;
- OMG Semantics of Business Vocabulary and Rules (SBVR) ориентирован на бизнес-ограничения;
- OMG Production Rule Representation (PRR) предназначен для описания правил для продукционных систем;
- W3C Rule Interchange Format (RIF) — семейство языков бизнес-правил для межсистемного взаимодействия.

Альтернативные подходы использованию бизнес-правил. Возможны различные подходы к использованию бизнес-правил при построении ИТ-систем. Можно выделить следующие основные подходы:

- бизнес-правила используются для формирования требований к системе на этапе проектирования;
- бизнес-правила «зашиваются» в программный код;
- используется интерпретатор бизнес-правил;
- бизнес-правила встраиваются в приложения, реализованные средствами языка высокого уровня, например Java;
- бизнес-правила встраиваются в BPEL-файлы.

Если в организации, занимающейся разработкой ИТ-систем, имеется репозитарий бизнес-правил, то разработчики систем могут использовать его при формировании требований к разрабатываемым ИТ-системам. После того как требования сформулированы в терминах бизнес-правил, то в процессе проектирования конкретной системы оно трансформируется в некоторое другое описание, например в UML-описание.

В ряде случаев естественным является описание функционирования системы в терминах правил. Например, если требуется разрабо-

тать программу для проверки знания таблицы умножения учащимися младших классов, то можно очень быстро написать такую программу, например на языке C++, и она будет служить много лет.

Прямо противоположный подход состоит в том, чтобы правила не зашивать в код, а хранить их в виде строк в отдельном файле и интерпретировать их с помощью интерпретатора. Подобный подход реализуется при построении экспертных систем, таких как CLIPS [23]. Пользователь или разработчик, желающий создать экспертную систему, приобретает оболочку и самостоятельно или с помощью эксперта наполняет ее правилами, которые затем обрабатываются интерпретатором, входящим в состав оболочки. В процессе эксплуатации правила можно легко добавлять или изменять. Возможность приобрести оболочку освобождает разработчика от необходимости разрабатывать собственный интерпретатор. Использование подобных систем ориентировано, прежде всего, на создание систем, использующих только правила.

Можно выделить смешанные подходы, когда модули, отвечающие за работу с правилами, встраиваются в код, написанный на Java или в код, описывающий бизнес-процесс.

Примером системы, позволяющей совместно использовать Java-код и правила, является система Jess [39].

В качестве примера систем, позволяющих использовать правила в системах управления бизнес-процессами, можно привести такие продукты, как ILOG JRules и JBoss Drools [28, 40].

Системы управления бизнес-правилами. Под системами управления бизнес-правилами (Business Rule Management System, BRMS) обычно понимают ИС, предназначенные для поддержки и выполнения бизнес-правил организации.

Типовая структура управления бизнес-правилами включает (рис. 6.5):

- среду разработки бизнес-правил;
- сервер исполнения бизнес-правил;
- подсистему тестирования бизнес-правилами;

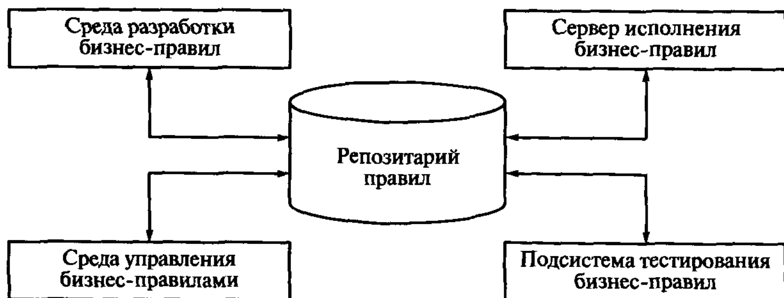


Рис. 6.5. Типовая структура управления бизнес-правилами

• среду управления бизнес-правилами;
• репозитарий бизнес-правил.

Среда разработки бизнес-правил представляет собой графическую среду, предназначенную для разработки и публикации бизнес-правил. Обычно это среда поддерживает режим групповой работы. Среда разработки ориентирована на разработчиков и интеграторов, но не на бизнес-пользователей.

Сервер исполнения бизнес-правил (Business Rule Engine), называемый иногда движком исполнения бизнес-правил, представляет собой компонент системы управления бизнес-правилами, в функции которого входит выполнение (интерпретация) правил. Можно считать, что сервер исполнения бизнес-правил предоставляет сервисы принятия решений, которые могут вызывать внешние приложения.

Типовой сервер исполнения бизнес-правил реализует следующие основные функции:

• исполнение правил;
• горячее развертывание правил;
• публикацию наборов правил, в частности в виде Web-сервисов;
• управление наборами правил;
• сбор статистики по выполнению правил.

Подсистема тестирования бизнес-правил позволяет разработчикам выполнять валидацию бизнес-правил.

Среда управления бизнес-правилами в отличие от среды разработки предназначена для бизнес-пользователей, не являющихся ИТ-специалистами. Это может быть менеджер, отвечающий за продажи.

Репозитарий бизнес-правил представляет собой хранилище бизнес-правил, доступ к которому осуществляется через среду разработки и среду управления. Обычно поддерживается работа с версиями.

В самом общем виде функционирование системы управления бизнес-правилами выглядит следующим образом.

На этапе разработки при формулировании требований к ИТ-системе определяются возможности системы по настройке парамеров функционирования в терминых бизнес-политик. Например, при разработке системы управления продажами одним из требований может являться обеспечение возможности для конечного пользователя системы (менеджера) реализовывать политику типа «Покупатель, который совершил единовременную покупку на определенную сумму, должен быть переведен в более высокую категорию».

Для того чтобы бизнес-пользователи могли редактировать и писать новые правила, на этапе разработки создается словарь бизнес-правил. Этот процесс иногда называют ***вербализацией*** (verbalization). Вербализация предполагает создание объектной бизнес-модели (business object model, BOM), которая может строиться на основе, например объектной модели, определенной в Java-проекте.

Бизнес-политика представляет собой совокупность нескольких правил. Бизнес-правила являются формализацией бизнес-политики,

представленной в форме последовательности «if-then» утверждений, например:

- if the customer's category is Silver and the value of the customer's shopping cart is more than $500;
- then change the customer's category to Gold;
- if the customer's category is Gold and the value of the customer's shopping cart is more than $1 000;
- then change the customer's category to Platinum.

После того как правила разработаны, они компонуются в исполняемый модуль и его можно вызывать как единую сущность.

Если используется СОА, а бизнес-процессы реализуются на языке BPEL, сервер исполнения бизнес-правил представляет собой сервис-принятия решений, на вход которого поступает SOAP-послание.

Каждое обращение к серверу исполнения бизнес-правил можно представить как выполнения одного или нескольких условных операторов. Если менеджер в некоторый момент времени решит, что для получения платиновой карты достаточно совершить покупки только на $700, то он может откорректировать эту цифру через доступную ему среду управления бизнес-правилами. Обычно это делается с помощью графического конструктора. Перед пользователем появляются список правил, отдельные поля которых он может редактировать, при этом не требуется не только перерабатывать код, но даже не требуется перезагружать систему.

Если по результатам эксплуатации системы появится идея дополнить политику еще одним правилом, в соответствии с которым покупателю, имеющему серебряную карту и истратившему $1 200, сразу выдается платиновая карта, то пользователь открывает конструктор правил и с помощью одного из доступных ему шаблонов создает новое правило посредством выбора значений элементов нового правила из выпадающего списка, что также можно сделать без перепрограммирования системы.

Для конкретных систем данный процесс может отличаться, однако принципиальным является то, что бизнес-пользователь может в определенных пределах изменять поведение системы без внесения изменений в код.

Практически все системы управления бизнес-правилами обеспечивают доступ к репозитарию бизнес-правил не только через GUI, но и позволяют делать это с помощью API. В этом случае можно добавлять правила, полученные автоматически, например с помощью систем извлечения знаний.

Использование бизнес-правил потенциально позволяет достичь следующих преимуществ:

- уменьшения времени разработки;
- оперативной реакции на изменения требований;
- возможности повторного использования кода;
- снижения стоимости разработки и владения ИТ-системой.

6.6. Порталы и портлеты

6.6.1. Порталы

Порталом (от лат. *porta* — ворота) принято называть Web-приложение, которое предоставляет пользователю Интернета или Интранета доступ к различным сервисам. Часто термин «портал» определяет единую точку доступа пользователя в информационное пространство, при этом предполагается, что пользователь, зайдя на портал, может получить доступ ко всем необходимым для него источникам информации и приложениям, поэтому порталы иногда определяют как рабочий стол (десктоп) нового поколения.

В процессе работы пользователи, как правило, имеют дело с данными разных форматов и происхождения и используют для их обработки различные приложения. Рабочее место, кроме того, предоставляет средства для интеграции людей и, в частности, средства для коллективной работы. Типовое портальное решение обеспечивает пользователя средствами поиска, обеспечением безопасности, доступом к системам электронного обучения и корпоративного документооборота.

По своей идее портал — это Web-сайт, ориентированный на удовлетворение информационных потребностей определенной категории пользователей. При этом каждый пользователь может настраивать портал под свои собственные нужды и может получать доступ к нему с помощью любого устройства, имеющего доступ в Интернет.

С точки зрения пользовательского интерфейса портал представляет собой многооконное интегрированное приложение. Каждое окно — это «зона», изображение в которой формируется отдельным приложением. За каждую зону отвечает отдельное приложение, однако приложения могут связываться между собой как автоматически, так и по требованию пользователя, при этом информация может синхронизироваться.

Часто разработчики называют порталами многостраничные сайты. Следует отметить, что не всегда бывает просто провести границу между порталом и сайтом, однако можно выделить, по крайней мере, три существенных различия между сайтом и порталом.

Во-первых, портал многофункционален. Титульная страница портала содержит все или почти все, что нужно для работы конкретного пользователя.

Во-вторых, портал предоставляет пользователю преимущественно динамический контент, который генерируется активными программными компонентами, имеется возможность синхронизации и обмена контентом между приложениями.

Во-третьих, портал может быть персонализирован, т.е. состав и внешний вид рабочего места зависят от роли пользователя и, кроме того, может быть изменен и настроен для удобства работы конкретного пользователя или группы пользователей.

Типовой портал кроме возможности одновременной работы с несколькими приложениями обеспечивает пользователю доступ к ряду сервисов общего назначения, таких как:

• сервис однократной регистрации, обеспечивающий аутентификацию пользователя при работе с порталом;

• сервис настройки и персонализации, позволяющий настраивать внешний вид, функциональность и информационное наполнение портала для нужд конкретного пользователя с учетом его роли;

• сервисы доступа к приложениям (сервисы обработки);

• сервисы доступа к бизнес-процессам, позволяющие пользователю быть участником бизнес-процесса в соответствии с определенной ему ролью;

• сервисы доступа к данным, обеспечивающие подключение пользователя к источниками консолидированной информации, такими как СУБД или хранилища данных;

• сервис поиска информации в Интернете и Интранете;

• сервис совместной работы, позволяющий пользователям работать в команде для решения общей задачи (разделяемые библиотеки документов, доски объявлений, онлайновые конференции, новостные группы);

• сервис публикации, позволяющий пользователю сохранять документы в хранилище контента портала;

• сервис подписки, который позволяет пользователю оформлять подписку и получать уведомления об изменении или появлении новой информации. При этом правила доставки/извещения и фильтрации могут настраиваться пользователем;

• сервис администрирования, позволяющий управлять правами доступа пользователей, создавать и удалять пользователей.

Большинство порталов представляют собой Web-порталы, работающие по принципу тонкого клиента, рабочим окном которых является окно Web-браузера.

С точки зрения назначения принято выделять три основных типа порталов:

• горизонтальные;

• вертикальные;

• корпоративные.

Горизонтальные, или общедоступные, порталы ориентированы на самую широкую аудиторию. Обычно контент таких порталов носит общий характер. Как правило, это новостная информация, рассылки, электронная почта и т. п.

Горизонтальные порталы имеют много общего со средствами массовой информации и могут рассматриваться как порталы общего назначения. В качестве примеров горизонтальных порталов могут выступать такие известные порталы, как Rambler, Lycos, Excite, Yahoo!

Вертикальные порталы можно рассматривать как специализированные порталы, предназначенные для информационного об-

служивания конкретных групп пользователей. В качестве примеров вертикальных порталов могут служить порталы B2C (Business-to-consumer), B2B (business-to-business), а также порталы типа B2E (Business-to-employees), которые обычно называют корпоративными порталами.

Примерами B2C-порталов могут служить порталы турфирм, предоставляющие такие услуги, как заказ билетов, бронирование мест в гостиницах и т.п.

B2B-порталы позволяют пользователям клиентам реализовывать совместные бизнес-операции, такие как выбор поставщиков, закупку товаров, проведение и участие в аукционах и т.д. Число B2B-порталов растет быстрыми темпами.

B2E-порталы, которые обычно называют корпоративными порталами, они предназначены для сотрудников, клиентов и партнеров одного предприятия.

Пользователи корпоративных порталов обычно получают доступ к сервисам и приложениям в зависимости от роли, назначенной конкретному пользователю.

Основным назначением корпоративного портала является предоставление внешним и внутренним пользователям возможности персонифицированного доступа к сервисам, приложениям и корпоративным данным — объединение изолированных моделей бизнеса, интеграция различных корпоративных приложений, включая приложения бизнес-партнеров, обеспечение доступа как стационарных, так и мобильных пользователей к корпоративным ресурсам независимо от местонахождения пользователя.

Следует отметить, что корпоративные порталы развиваются быстрыми темпами. Обычно принято выделять четыре поколения корпоративных порталов [53].

Корпоративные порталы первого поколения ориентированы на предоставление пользователю преимущественно статического Web-контента и Web-документов.

Обычно целью разработки таких порталов является создание единой точки доступа к корпоративной информации, распределенной по разным подразделениям организации. Такие порталы, как правило, реализует следующий типовой набор функций:

- персонализация;
- системы поиска информации;
- управление контентом и его агрегация;
- наличие средств и инструментов интерации приложений.

В ***корпоративных порталах второго поколения*** появляются такие черты, как наличие персонализации контента, присутствие поисковика. Эти порталы ориентированы на использование в качестве составной части КИС, и для них характерны:

- надежная среда реализации приложений;
- мощные инструменты разработки и интеграции приложений;

• соответствие требованиям, предъявляемым к ИС масштаба предприятия;

• поддержка интеграции с ИС партнеров;

• наличие поддержки мобильного доступа к ресурсам.

Корпоративные порталы третьего поколения ориентированы преимущественно на предоставление сервисов в отличие от порталов первого и второго поколений, которые ориентированы на предоставление контента. Отличительной чертой порталов третьего поколения является то, что в них реализуется идея сотрудничества (collaboration). Порталы третьего поколения предоставляют возможность сотрудникам работать в виртуальном офисе и предоставляют такие возможности как чаты, e-mail, возможность групповой работы над проектами. Порталы третьего поколения — это преимущественно корпоративные порталы.

Для *четвертого поколения корпоративных порталов* характерны:

• ориентация на электронный бизнес, что подразумевает интеграцию с модифицируемыми, переносимыми в новое окружение приложениями;

• возможность работы не только с сервисами, но и с политиками;

• пользователям предоставляется возможность получать доступ к информации с помощью многих типов устройств, в частности мобильных устройств.

Современный корпоративный портал обычно представляет собой продукт или набор продуктов, который базируется на определенной

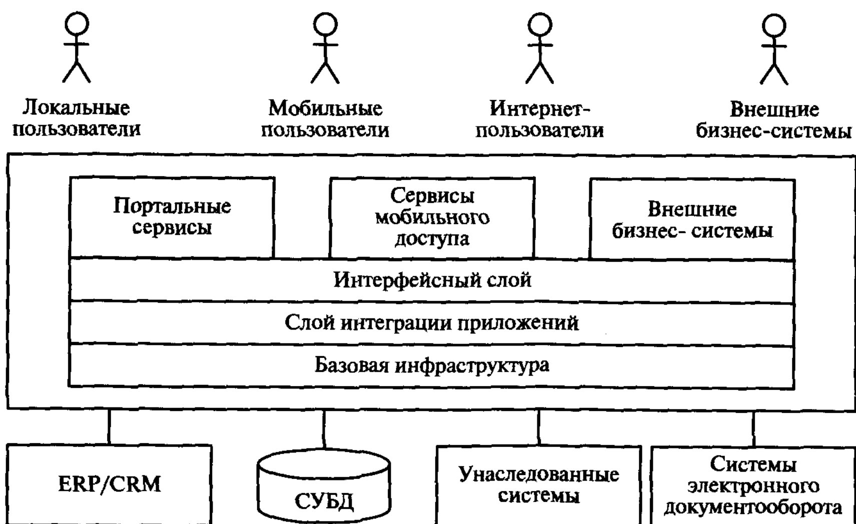


Рис. 6.6. Состав типовой КИС, ориентированной на использование порталов

инфраструктуре, в качестве которой, как минимум, выступают серверы приложений и серверы баз данных.

В составе типовой КИС, ориентированной на использование порталов можно выделить три основных функциональных слоя (рис. 6.6):

- базовая инфраструктура;
- слой интеграции приложений;
- интерфейсный слой.

Базовая инфраструктура отвечает за предоставление таких базовые сервисов, как управление пользователями, управление безопасностью, управление транзакциями др.

Слой интеграции приложений обеспечивает взаимодействие портала с приложениями, такими как ERP- и CRM-системы, унаследованные приложения, СУБД и др.

К интерфейсному слою принадлежат визуальные и невизуальные компоненты порталов, называемые обычно портлетами. Интерфейсный слой, включающий в себя средства управления информационным наполнением, адаптеры для обмена данными с информационными системами бизнес-партнеров, интерфейсные модули для поддержки взаимодействия с мобильными устройствами и др.

6.6.2. Портлеты

Информационное наполнение порталов часто предоставляется пользователям в форме портлетов-контейнеров. С точки зрения реализации, портлет представляет собой фрагмент кода, исполняемый на портальном сервере. Когда говорят о портлетах, то речь ведется в терминах JEE, поскольку эта платформа поддерживает работу с портлетами, хотя порталы с успехом можно строить и на других платформах.

Портлет — это приложение, предоставляющее пользователю некоторую информацию или сервис. Он может быть включен в качестве составной части в портальную страницу. Портлет работает под управлением контейнера, который обрабатывает запросы и генерирует динамический контент, и представляет собой plugin, ответственный за представление информации пользователю.

Динамически сгенерированный контент портлета называют фрагментом. Контент нескольких портлетов можно объединять в рамках одной портальной страницы.

Типовая структура портальной страницы показана на рис. 6.7.

Клиент, обычно это Web-клиент, взаимодействует с портлетом в режиме запрос-ответ. Для разных пользователей портлет может иметь разный внешний вид в зависимости от настроек.

Портлеты могут реализовываться на разных платформах. При реализации на Java-платформе портлет рассматривается как Web-компонент.

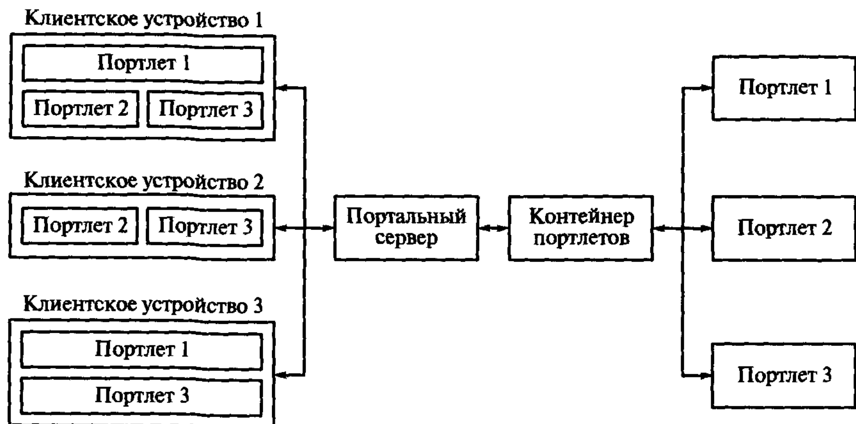


Рис. 6.7. Типовая структура портальной страницы

На данный момент рабочей является версия 2.0 и дальнейшее изложение ведется применительно к данной спецификации.

Контейнер портлетов представляет собой среду, в которой «живут» портлеты. Управляя жизненным циклом портлетов, контейнер не отвечает за агрегацию портлетов.

Обычно портал или точнее портальный движок и контейнер могут реализовываться как один монолитный компонент или как два независимых компонента.

В самом общем виде работа с портальной страницей происходит следующим образом:

- Web-клиент посылает HTTP запрос к порталу;
- портал получает запрос и определяет, к какому из портлетов, относящихся к данной портальной странице, относится запрос;
- портал запускает портлет через контейнер и получает требуемый контент;
- портал агрегирует полученный контент в портальную страницу и отправляет ее клиенту.

Следует заметить, что портлеты имеют много общего с сервлетами, в частности:

- как сервлеты, так и портлеты, являются Web-компонентами;
- подобно сервлетам портлеты работают под управлением контейнера;
- портлеты подобно сервлетам генерируют динамический контент;
- клиент работает через Web-интерфейс.

Отличие портлетов от сервлетов состоит в следующем:

- портлет представляет собой только часть изображения, портальная страница формируется из фрагментов средствами портального сервера;

• Web-клиент взаимодействует с портлетом не напрямую, а через портальную систему;

• форматом отображения портлета можно управлять;

• на одной портальной странице один и тот же портлет может появляться несколько раз.

Кроме того, портлеты по сравнению с сервлетами, обладают следующей дополнительной функциональностью:

• портлеты работают с конфигурационными файлами;

• портлеты имеют доступ к профилям пользователей;

• портлеты могут сохранять свое состояние;

• портлеты могут взаимодействовать друг с другом и контейнером посредством сообщений.

Контейнер портлетов можно рассматривать как расширение Web-контейнеров.

Портлет генерирует фрагмент в форме гипертекста. Портал формирует окно портлета посредством добавления к сгенерированному фрагменту обрамления, включающее рамку и кнопки. Затем окна портлетов агрегируются в портальную страницу (рис. 6.8).

Для формирования портлетов могут использоваться либо JSP, либо такие фреймворки, как Java Server Faces (JSF), Struts, Spring [21] и др.

Портлеты могут функционировать в нескольких режимах. Пользователь может работать с информационным наполнением, может настраивать внешний вид портлета в соответствии со своими предпочтениями, а администраторы могут конфигурировать портал для предоставления. Режим, который пользователи выбирают, определяет, какой интерфейс портлета они видят. Представление может находиться в одном из следующих состояний: нормальное (normal), максимизированное (maximized), минимизированное (minimized), закрытое (closed).

Свойства, существенные для размещения портлетов, хранятся в дескрипторе.

Сервер порталов предоставляет портлету сервис получения информационного наполнения и сервис сохранения состояния между сеансами, а также сервис доступа к пользовательской информации,

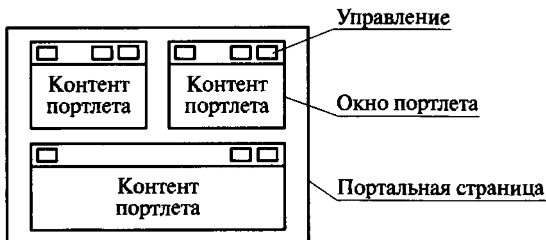


Рис. 6.8. Формирование портальной страницы

которая дает портлетам доступ к информации, касающейся пользователей, в том числе предпочтения пользователя, данные о настройке и т.д.

API портлетов определяет интерфейс между портлетом и контейнером портлетов.

Концепция порталов создавалась постепенно в течение достаточно длительного промежутка времени. Еще за много лет до появления самого терминов «портал» и «портлет» разработчики сайтов широко использовали такие приемы, как введение динамического контента, фреймы, элементы персонализации страницы. Однако различные Web-серверы и серверы приложений обеспечивали эти возможности разными способами. С появлением концепции портала появилась необходимость стандартизации средств построения портальных приложений. В рамках технологии JEE в 2003 г. появилась спецификация JSR-168, которая была разработана совместно фирмами IBM и Sun Microsystems. Она определяла портлеты, написанные на языке Java.

В 2007 г. появилась вторая версия данной спецификации под номером JSR-286 [44], которая является расширением версии 1. Обе версии совместимы на двоичном уровне и это означает, что все портлеты, созданые в соответствии со спецификацией JSR-168, могут работать в контейнере, созданном в соответствии со спецификацией JSR-286.

На практике портлеты, созданные в соответствии со специцификацией JSR-168, продолжают активно использоваться.

Функции для работы с портлетами определены в пакете javax.portlet. API портлетов определены таким образом, чтобы можно было провести четкую границу между портлетом и портальной инфраструктурой.

Хотя API портлетов во многом похожи на API сервлетов, но имеются и отличия, в частности, контейнер портлетов различает и позволяет поддерживать как состояние портлета, так и состояние окна. Состояние окна определяет, какую часть экранной страницы занимает данный портлет. Окно может находиться в одном из трех состояний: нормальном, минимизированном или максимизированном.

В соответствии со спецификацией JSR-168 определяются следующие методы интерфейса Portlet: init(), processAction(), render(), destroy().

Метод init() используется для инициализации портлета.

Метод processAction() вызывается для обработки действий пользователя в портлете.

Метод render() вызывается при необходимости перерисовки изображения портлета.

Метод destroy() вызывается перед удалением портлета из памяти.

Портлет может находиться в одном из трех состояний:

- View (Представление);

- Edit (Редактирование);
- Help (Помощь).

В состоянии «Представление» портлет реализует свою функциональность, в состоянии «Редактирование» реализуются функции, связанные с настройка портлета, а в состоянии «Помощь» отображаются подсказки.

Наряду с интерфейсом Portlet в пакете определяется класс GenericPortlet, где метод render() определен таким образом, что управление в зависимости от текущего состояния портлета передается одному из методов: doView(), doEdit() или doHelp(). Большинство реальных портлетов наследуют от класса GenericPortlet.

Ключевым при работе с портлетом безусловно является метод processAction(), который работает с объектами типа запрос и ответ — объекты ActionRequest и ActionResponse соответственно. Для метода render() и вызываемых им методов doXxx() в качестве аргументов и результатов выступают объекты RenderRequest и RenderResponse. Используя эти объекты, портлет может управлять состоянием и взаимодействовать с другими портлетами, сервлетами, принимать данные, вводимые пользователем в экранные формы портлета, создавать изображение для отображения в портале и посылать его клиенту и определять состояние портлета.

Запрос клиента инициируется при помощи ссылок URL, создаваемых портлетом. URL портлета могут быть двух типов: URL действия и URL перерисовки. Портлет создает свои URL, вызывая методы createActionURL и createRenderURL интерфейса RenderResponse. Эти методы возвращают интерфейс PortletURL. URL перерисовки может быть уточнен указанием состояния портлета, для которого этот URL применяется. Обычно запрос клиента, инициируемый URL-действия, превращается в один запрос действия и много запросов перерисовки — по одному на каждый портлет на странице портала. Если портлет может кешироваться (это определяется в дескрипторе портлета), метод render может не вызываться.

Портлеты — настраиваемые элементы. Настройки портлета сохраняются в виде наборов пар «имя — значение». За сохранение конфигурации портлета отвечает контейнер. Программист работает с настройками посредством объекта типа PortletPreferences, который имеет методы setValue() и getValue() установки (изменения) значений настроек и чтения их значений соответственно.

Портлеты упаковываются как стандартные в JEE Web-архивы (файлы WAR). Кроме дескриптора развертывания web.xml они дополнительно содержат дескриптор портлета — portlet.xml, в котором определены элементы конфигурации портлета.

Основными средствами обеспечения безопасности портлетов в рамках спецификации JSR-168 являются возможность установки в дескрипторе портлета флажка, требующего, чтобы портлет работал только через защищенный протокол HTTPS и возможность аутенти-

фикации пользователей и ролей. При этом не определяется, каким образом реализованы пользователи и их роли.

Кроме того, определяется библиотека тегов JSP, помогающая отображать портлет при помощи технологии Java Server Pages. При реализации портлетов довольно часто в методе, например doView(), выполняется логика, связанная с анализом состояния, например некоторая бизнес-логика, а для отображения портлет вызывает страницу JSP, формирующую код фрагмента HTML-страницы.

При вызове страницы JSP ей передаются запрос и отклик портлета. Перед вызовом страницы портлет может сохранить какие-то объекты как атрибуты запроса.

Страница может направлять запрос в портлет, определяя адрес портлета через теги специальной библиотеки.

За агрегацию фрагментов, поставляемых разными портлетами в полную страницу портала, отвечает контейнер портлетов.

Стандартизация портлетов (как и само осознание концепции порталов) несколько задержалась. На момент создания спецификации уже существовало большое число реализаций серверов порталов, в которых использовались фирменные средства и собственные API. Спецификация JSR-168, однако, была принята производителями серверов порталов, и в настоящее время большинство продуктов этого класса проходят или уже прошли процесс адаптации к стандарту. Вместе с тем эти продукты вынуждены поддерживать и свои старые API, так что различия в диалектах API быстро исчезнуть не могут.

Следует отметить, что спецификация JSR-168 определяет только базовые средства работы с портлетами. Обычно развитые портальные серверы предлагают многочисленные и не предусмотренные спецификацией дополнительные возможности. При этом у каждого из производителей имеются собственные расширения для работы с портлетами.

В 2007 г. в рамках Java Community Process была опубликована окончательная вторая версия спецификации портлетов (JSR-286), которая дополняет предыдущую спецификацию, обеспечивая стандартизацию возможностей, не охваченных в JSR-168. В качестве новых элементов, появившихся в рамках спецификации JSR-268, можно выделить следующие:

- обеспечение механизмов обмена событиями между портлетами: портлет может объявлять события, которые он хочет генерировать, и события, которые он хочет получать; контейнер работает как диспетчер событий;
- поддержка работы с фильтрами, которая выражается в возможности преообразовывать «на лету» как входную, так и выходную информацию;
- обеспечение возможности для портлетов совместно использовать общий сеанс (сеанс пользователя) и данные, относящиеся к общему сеансу;

• обеспечение поддержки работы фреймворками Java AJAX, Server Faces, Struts и др.

Обычно портал получает информационное наполнение для своих портлетов из многих как внутренних, так и внешних источников. Для того чтобы интегрировать новый источник, администратор портала должен адаптировать информационное наполнение к формату, который воспринимается порталом, что может оказаться весьма длительным и трудоемким процессом.

Возможны два альтернативных подхода к решению данной задачи.

Первый подход предполагает, что в каждом случае создается портлет, обеспечивающий пользовательский интерфейс для выполняемой сервисом функции: ввод параметров запроса и представление результатов. На основании полученных данных портлет получает требуемые данные, например из БД, либо формирует SOAP-запрос к сервису и превращает отклик сервиса в графическое экранное представление. Все портлеты работают на одном портальном сервере. Подобный подход требует написания кода портлета, обеспечивающего пользовательский интерфейс, извлечение данных и развертывания портлета. Альтернативный подход заключается в том, что удаленный Web-сервис сам будет генерировать фрагмент разметки страницы, который непосредственно будет включаться в страницу нашего портала.

Второй подход предполагает использование специальных посредников, использование которых позволяет провайдерам поставлять контент без ручных настроек (рис. 6.9).

Идея состоит в том, что вместо добавления самого портлета в портальный сервер туда помещают его заместителя, который взаимодействует с удаленным портлетом.

Сам удаленный портлет поддерживается другим сервером порталов или модулем исполнения портлетов. Последний представляет собой упрощенный сервер порталов (среда исполнения портлетов), который дает возможность удаленному портлету реагировать на вызовы посредника. Он может быть реализован в другой технологии,

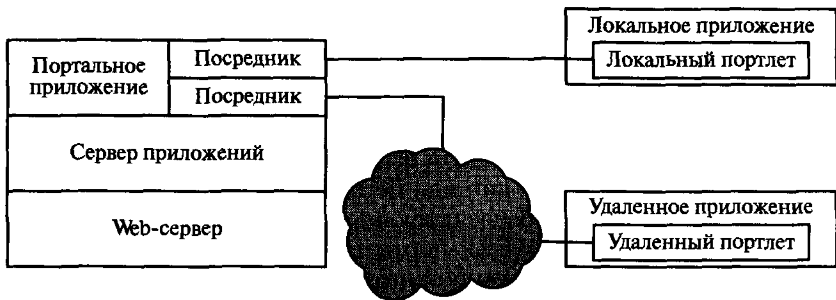


Рис. 6.9. Процесс создания портлета

например .Net, но при этом должен поддерживать протокол удаленного вызова портлета.

Для обеспечения совместимости между портальными серверами разных производителей и поставщиками информационного наполнения требуется стандартизованная модель взаимодействий для посредника портлета и удаленного портлета.

Подобный стандарт создан в рамках организация Organization for the Advancement of Structured Information Standards (OASIS) — Web-службы для удаленных порталов (Web services for remote portals, WSRP).

Спецификация WSRP является продуктом международной организации Organization for the Advancement of Structured Information Standards (OASIS). Организация по развитию стандартизации структурированной информации является консорциумом, содействующим принятию технических стандартов. Версия 1.0 спецификации WSRP была закончена в 2003 г., а версия 2.0 появилась в 2008 г. [60].

Спецификация WSRP определяет интерфейс для взаимодействия с подключаемыми, ориентированными на представление Web-сервисами, которые обеспечивают взаимодействие с пользователями и выступают в качестве поставщиков фрагментов кода на языке разметки для агрегирования в портальные страницы. WSRP представляют собой визуальные компоненты, которые можно подключать с использованием технологии визуального проектирования. В определенном смысле удаленные портлеты, обладающие графическим интерфейсом, можно рассматривать как Web-сервис, и для их описания может использоваться WSDL.

WSRP работает следующим образом. Поставщики информационного контента реализуют свой сервис как Web-сервис удаленного портала и публикуют ее, например в UDDI.

Можно привести, по крайней мере, два довода в пользу создания отдельного стандарта Web-сервисов для портлетов.

Во-первых, обычные Web-сервисы ориентированы на обработку запросов и генерацию откликов, выполняемых преимущественно на программном уровне, в то время как портлеты ориентированы на работу с графическим интерфейсом.

Во вторых, требуется четко определенный интерфейс, определяющий то, как портал взаимодействует с сервисом и собирает фрагменты разметки в портальную страницу. Следует заметить, что удаленные и локальные портлеты могут работать на одном портале и для конечного пользователя они неразличимы.

Работа с удаленным портлетом выглядит следующим образом.

Поставщик предлагает один или несколько портлетов и реализует ряд интерфейсов WSRP, обеспечивающих общий набор операций для конечных пользователей. WSRP-портлет представляет собой подключаемый компонент, формирующий интерфейс конечного пользователя. Он выполняется внутри поставщика WSRP и доступен удаленно через интерфейс, определенный этим поставщиком.

Потребитель WSRP представляет собой клиента Web-сервиса, который вызывает WSRP-сервис и обеспечивает среду взаимодействия пользователя с портлетами, предлагаемыми одним или несколькими поставщиками; обычно роль потребителя WSRP играет портал.

Реально Web-сервисом является не WSRP-портлет, а поставщик WSRP, именно он имеет стандартное описание на языке WSDL и набор конечных точек. Обращение к WSRP-портлету возможно только через поставщика. Поставщик принимает SOAP-запрос, выделяет из него обращение к портлету и упаковывает фрагмент разметки, генерируемый портлетом, в ответное SOAP-сообщение. В функции же потребителя WSRP входит упаковка в SOAP-запрос параметров формы, поступившей от пользователя, и выделение из SOAP-отклика фрагмента разметки, присланного поставщиком и WSRP-портлетом.

В рамках спецификации WSRP определены два обязательных и два необязательных интерфейса, которые должны реализовывать поставщики WSRP и использовать потребители WSRP для взаимодействия с удаленными портлетами. Стандартизация этих интерфейсов дает возможность потребителю WSRP обращаться к любому поставщику.

Обязательными для реализации интерфейсами WSRP являются интерфейс описания сервиса (Service Description Interface) и интерфейс разметки (Markup Interface).

Через *Интерфейс описания сервиса* потребители могут запрашивать, какие портлеты предлагает поставщик, и получать информацию о самом поставщике.

Интерфейс разметки предназначен для поддержки взаимодействия поставщика с удаленными портлетами. Он позволяет посылать им формы от пользователя портальной страницы и получать от них информацию о текущем состоянии портлета.

Необязательными интерфейсами WSRP являются интерфейс регистрации (Registration Interface) и интерфейс управления портлетом (Portlet Management Interface).

Интерфейс регистрации предназначен для поддержки процесса регистрации потребителя перед тем, как тот сможет обратиться к сервису. Это позволяет поставщику сервиса адаптировать свое поведение применительно к определенному типу потребителя; через этот интерфейс поставщик и потребитель могут обмениваясь сведениями о себе.

Интерфейс управления портлетом позволяет пользователю управлять жизненным циклом удаленного портлета и настроить его поведение.

Все перечисленные выше интерфейсы WSRP представляют собой XML-протоколы и соответственно платформенно независимы.

Следует особо отметить, что WSRP не заменяет стандарты Web-сервисов, а дополняет их и предствляет собой надстройку над ними.

Все взаимодействия между потребителем и поставщиком WSRP осуществляются по протоколу SOAP, а операции интерфейсов WSRP определяются в WSDL-описании WSRP. Использование подобного подхода позволяет осуществлять публикацию WSDL-описаний WSRP-портлетов в реестрах UDDI.

Типовой сценарий использования WSRP на основе технологии UDDI выглядит следующим образом.

1. Провайдер разрабатывает набор портлетов, назначая WSRP-поставщика и отображая их как WSRP-портлеты. Если провайдер хочет, чтобы эти портлеты использовались как удаленные, то он публикует их описания в реестре.

2. Пользователь, заинтересованный в работе с удаленными портлетами, ищет нужный ему портлет, используемые предоставляемые порталом средства, либо независимое приложение.

3. После обнаружения желаемого портлета пользователь добавляет новое приложение к одной из своих портальных страниц.

4. Если пользователю не разрешено добавление портлетов к своей странице, то администратор портала находит их в UDDI-реестре и сделает их доступными для пользователей, добавив во внутренний реестр портала.

5. После того как удаленный портлет помещен на портальную страницу пользователя, он увидит на своей портальной странице выполняющийся удаленно портлет, при обращении к которому портальный сервер выполняет запрос к соответствующему Web-сервису посредством отправки SOAP-послания, а в ответ получает SOAP-послание с фрагментом на языке разметки страниц, который портал интегрирует в отображаемую страницу. При этом конечный пользователь полностью избавлен от необходимости знать детали работы WSRP.

Как у всякой технологии, у технологии портлетов имеются свои области применения достоинства и недостатки.

Использование портлетов безусловно целесообразно в случае, если цель проекта состоит в том, чтобы объединить Web-приложения и информацию в одном удобном месте, если требуется использовать сервисы внешних поставщиков и (или) выступать в качестве поставщика сервиса. Если пользователи активно перемещаются и пользуются мобильными устройствами, то портлеты — очевидный выбор.

Если цели проекта отличаются от указанных выше, то следует рассмотреть другие альтернативы.

Достоинства портлетов:

• возможность работать на различных клиентских устройствах, что позволяет пользователям перемещаться с компьютера на компьютер и с одного мобильного устройства на другое, при этом использовать ту информацию и те приложения, которые им нужны;

• внешний вид и функциональность портлетов можно настраивать для различных групп пользователей, а сами пользователи мо-

гут настраивать внешний вид портлетов в соответствии со своими предпочтениями;

• выполнение портлетов как Web-сервисов в целях обеспечения доступа к ним внешних пользователей;

• разделение сложных приложений на отдельные задачи по принципу — одна группа тесно связанных задач представлена одним портлетом;

• возможность относительно легко добавлять новую функциональность к уже существующим приложениям;

• хорошая совместимость с брандмауэрами (firewalls).

Портлеты не могут быть решением для каждого проекта. Далее перечислены типовые ситуации, когда не следует использовать портлеты.

Недостатки портлетов:

• сложные пользовательские интерфейсы плохо переводятся в портлеты, использующие такие языки разметки, как HTML и WML;

• если требуется реализовать пользовательские интерфейсы с часто обновляемыми данными, то использование портлетов может привести к замедлению работы системы, поскольку, когда обновляется один портлет, то перерисовывается вся портальная страница;

• высокоинтерактивные пользовательские интерфейсы недостаточно хорошо переводятся в Web-приложения, вообще, и в портлеты, в частности. Если необходимо, чтобы интерфейс приложения изменялся в процессе работы, например необходимо появление выпадающего меню, вы можете либо использовать форму и обновлять всю страницу полностью, либо использовать скрипты, чтобы изменить портлет. Однако всплывающие окна и скрипты обычно не могут использоваться на мобильных устройствах;

• имеются проблемы при работе с формами (внутренние фреймы разрешены, но только пользователи Microsoft Internet Explorer могут их видеть);

• портлеты полностью не стандартизированы, и они еще не поддерживаются на стольких платформах, как другие Java-технологии.

6.7. Корпоративные сервисные шины

6.7.1. Общие принципы построения

Основная задача интеграции приложений состоит в объединении функций двух или более приложений для получения новой функциональности.

Можно выделить два основных типа задач интеграции приложений:

• задачи интеграции корпоративных приложений;

• задачи интеграции приложений, функционирующих в составе КИС разных организаций.

Системы интеграции корпоративных приложений (Enterprise Applications Integration, EAI) — технологии, ориентированные на решение проблем интеграции различных систем, приложений и данных внутри отдельной организации. Иногда для этих технологий используется аббревиатура A2A (Application-to-Application — приложение-приложение).

Системы интеграции между организациями, которые обычно называют B2B (Business-to-Business Integration), представляют собой технологии, ориентированные на обеспечение безопасного, надежного информационного обмена между ИС различных организаций. Эти технологии направлены на автоматизацию бизнес-процессов в рамках «расширенных организаций», что создает предпосылки для создания разного рода виртуальных организаций.

Провести четкую границу между интеграцией типа A2A и B2B не всегда просто, поскольку в рамках крупной корпорации, даже если приложения работают в рамках одной организации, они могут быть разнесены территориально и находиться в разных автономных системах, быть отделены с помощью firewalls и подчиняться разным политикам безопасности.

Технологии и системы управления бизнес-процессами являются результатом естественной эволюции классических систем делопроизводства и документооборота (workflow systems), систем класса EAI и B2B. Ключевое отличие состоит в том, что если традиционные системы управления документооборотом предназначались преимущественно для обмена информацией между людьми, современные системы управления бизнес-процессами класса A2A ориентируются на обмен данными между приложениями, а системы класса B2B — на интеграцию данных в межведомственной среде, технологии BPM интегрируют данные, приложения и людей через единые бизнес-процессы.

Данный подход достаточно адекватно отражает современную точку зрения, состоящую в том, что основным предметом интеграции должны быть бизнес-процессы и, в частности бизнес процессы, которые «пересекают» границы различных приложений, подразделений и организаций.

Когда говорят об интеграционных решениях, то обычно выделяют три альтернативных подхода:

- точка-точка (point-to-point);
- шлюз (hub-and-spoke);
- шина (bus).

Топология интеграционного решения отражает различные способы взаимодействия приложений, среди которых можно выделить соединения точка-точка, шлюзы и шины, которые показаны на рис. 6.10, *а*, *б*, *в* соответственно.

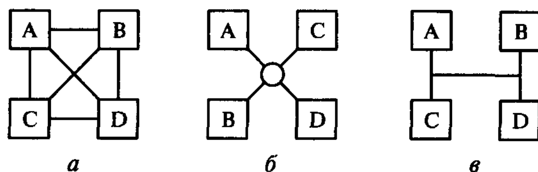


Рис. 6.10. Варианты интеграционных решений

Соединение типа точка-точка предполагает, что интегрируемые приложения устанавливают прямые связи друг с другом, при этом каждая из связей может быть реализована уникальным способом. Для реализации каждой связи может использоваться собственная технология. Очевидно, что это самый простой и в ряде случаев самый действенный подход к решению задачи интеграции приложений, когда требуется объединить 3-4 приложения. Однако по мере увеличения числа интегрируемых приложений связи между приложениями становятся многочисленными, что резко повышает трудоемкость управления ИС. Кроме того, при модификации одного из приложений приходится разрабатывать несколько новых адаптеров, что означает усложнение поддержки всей системы.

Основными недостатками интеграции приложений по принципу точка-точка являются следующие:

- недостаточная гибкость;
- сложность поддержки многочисленных соединений типа точка-точка;
- изменения в одном приложении сказывается на других приложениях;
- логика маршрутизации часто зашивается в код;
- часто отсутствует общая модель безопасности;
- используются разные API;
- используются протоколы принадлежащие разным стекам;
- сложности с созданием фреймворков;
- сложности с поддержкой асинхронных взаимодействий;
- сложности реализации бизнес-процессов с очень большим временем жизни;
- проблемы с мониторингом состояния;
- низкая надежность.

Переход от модели взаимодействия точка-точка к моделям взаимодействия на основе централизованного шлюза или на основе шины обусловлен существенным уменьшением количества интерфейсов. Модели интеграции на основе шлюза и шины относятся к классу программного обеспечения промежуточного уровня (middleware), обеспечивающего трансформацию, транспортировку, маршрутизацию данных.

Следует отметить, что с использованием термина EAI имеются некоторые особенности. В частности, данный термин употреблял-

ся выше в смысле EAI-технологии (технологии, ориентированные на реализацию интеграции типа A2A). До начала 2000-х гг. термином EAI обозначались также продукты, предназначенные для реализации данного типа интеграции. Однако в начале 2000-х гг. появился продукт нового поколения, который чаще всего стали называть корпоративной сервисной шиной (Enterprise Service Bus, ESB) [54], поэтому используя термины EAI и ESB, ряд авторов, в частности [33, 49, 54], рассматривают системы EAI как предшественников ESB, при этом указывают на два основных различия между ними.

Первое различие состоит в том, что EAI-системы построены по принципу шлюза и используют модель «вставь и говори» (hub-and-spoke), представляющие собой централизованную архитектуру, в которой весь обмен данными осуществляется через центральный хаб или брокер. ESB использует шинную модель с распределенной архитектурой, она может быть реализована как несколько отдельных распределенных подсистем.

Второе различие состоит в том, что в отличие от EAI ESB-системы ориентированы на применение открытых стандартов, таких как Java Message Service (JMS), XML, J2EE Connector Architecture (JCA) and Web-сервисы.

В последние годы появились такие новые ИТ-технологии как Service Oriented Architecture (SOA), Enterprise Application Integration (EAI), Business-to-Business (B2B) и Web services. Все они в значительной мере направлены на поддержку реализации интегрированных бизнес-процессов. Наилучшим образом организовать поддержку интегрированных бизнес-процессов можно с помощью ESB. ESB — это новый по отношению к EIA-интеграции подход, который представляет собой интеграционную платформу, позволяющую использовать различные механизмы интеграции:

- работу с очередями сообщений;
- Wеб-сервисы;
- преобразование данных;
- интеллектуальный роутинг.

ESB дает возможность интегрировать приложения и бизнес-процессы как внутри организации, так и между организациями.

Корпоративные ИС в большинстве своем представляют собой системы, управляемые событиями. События могут наступать в произвольный момент времени в произвольном порядке. Отдельные бизнес-службы также работают по принципу управления событиями, поэтому они могут работать независимо.

ESB представляет собой шину (backbone), которая работает по принципу слабосвязанной системы, управляемой событиями. Каждый источник данных представлен в форме конечных точек (endpoints).

ESB обеспечивает собой достаточно общий подход к интеграции, причем интеграцию можно реализовывать постепенно, добавляя отдельные элементы.

ESB первого поколения появились в 2002 г. и были поддержаны практически всеми ведущими поставщиками корпоративных решений. ESB второго поколения — это инфраструктура, поддерживающая достаточно широкий спектр технологий, в частности, работу с сообщениями и Web-сервисами, преобразование и маршрутизацию сообщений. ESB второго поколения появились и активно используются начиная с середины 2000-х гг.

Концепция ESB самым тесным образом связана с концепцией СОА. При этом ESB поддерживает концепции реализации СОА по принципу: представление службы отделяется от ее реализации.

Основные фукнции ESB:

- обеспечение интерфейсов взаимодействия;
- отправка сообщений и маршрутизация;
- преобразование данных;
- реакция на события;
- управление политиками;
- виртуализация.

Типовой список требования, предъявляемых к ESB со стороны пользователей, выглядит следующим образом:

- ESB должна иметь достаточную пропускную способность для того, чтобы обеспечивать одновременное функционирование большого числа соединений;
- ESB должна поддерживать несколько стилей интеграции, в частности, интеграцию, основанную на событиях, и интеграцию, ориентированную на сообщения;
- ESB должна позволять приложениям работать с сервисами как напрямую, так и через адаптеры, при этом должен иметься достаточный набор таких адаптеров, к числу которых могут быть отнесены следующие: адаптеры доступа к ERP-системам, адаптеры доступа к компонентам, адаптеры доступа к Java-объектам, адаптеры доступа к данным, адаптеры доступа к унаследованным системам.

Минимально необходимые возможности ESB:

- ESB представляет собой логический архитектурный компонент, который приводит интеграционную инфраструктуру в соответствие с принципами СОА и реализуется в виде распределенной гетерогенной инфраструктуры;
- интегрируемые приложения представляются в виде набора сервисов.

ESB реализует две основные фукнции: преобразование форматов сообщений и маршрутизацию сообщений.

Внедрение и адаптация шлюза на основе интеграционного сервера или шины корпоративных сервисов связаны с существенными капитальными вложениями. В то же время необходимость использования интеграционного решения класса middleware продиктована не только трудоемкостью управления моделью точка-точка, но и ее недостаточной надежностью.

Модель точка-точка может быть использована для системы с небольшим числом приложений на начальной стадии интеграции. С ростом количества информационных ресурсов необходимо использование интеграционного решения более высокого порядка, к которым относятся интеграционный сервер и шина корпоративных сервисов.

В качестве реализации модели интеграционного сервера можно назвать продукты класса EAI, объединяющие корпоративные приложения управления кадрами, управления отношениями с клиентами, бухгалтерии и т.д.

Модель шины реализуют продукты класса ESB, объединяющие Web-сервисы, которые разработаны на базе существующих корпоративных приложений.

Интеграционное решение EAI построено на централизованных принципах. Это означает, что все функции промежуточной обработки сообщений выполняются единым интеграционным сервером, включая трансформацию и перераспределение сообщений. Среди преимуществ EAI выделяется эффективность администрирования корпоративной системы, среди недостатков — высокая стоимость приобретения, не позволяющая предприятиям малого бизнеса использовать интеграционное решение данного класса.

Модель интеграции на базе шины корпоративных сервисов ESB представляет собой информационную инфраструктуру, построенную на принципах сервис-ориентированной архитектуры. Данные принципы предполагают, что объектами взаимодействия внутри предприятия являются Web-сервисы, представляющие функции приложений в виде автономных программных модулей. Все аспекты разработки и вызова Web-сервисов основываются на Web-стандартах XML и HTTP, что обеспечивает платформенно-технологическую независимость формата взаимодействия корпоративных приложений. Общим между моделями EAI и ESB является то, что они обеспечивают промежуточную обработку сообщений. Однако в модели ESB функции брокера и оркестровки сообщений являются техническими сервисами, которые могут происходить от разных поставщиков, быть физически распределенными и использовать разные технологии коммуникации. Несмотря на технические различия, продукты семейства ESB совместимы между собой, что обеспечивается политикой открытых интерфейсов (open interface policy). Независимость от поставщика позволяет применять широкий спектр возможностей приобретения и аренды необходимого программного обеспечения. Это обеспечивает высокую степень адаптируемости, сосуществования, совместимости и взаимозаменяемости элементов ESB.

Распределенная архитектура ESB характеризуется более сложным управлением и администрированием, но является более гибкой и масштабируемой. Кроме того, внедрение сервис-ориентированного

подхода при взаимодействии корпоративных приложений не требует изменений во всех элементах системы и может происходить поэтапно, интегрируя лишь критические для бизнеса приложения.

6.7.2. Обобщенная архитектурная модель интеграционной подсистемы

ESB можно рассматривать как многоуровневую (многослойную) систему, при этом можно выделить пять уровней:

- уровень сопряжения (адаптеры и интерфейсы);
- транспортная подсистема;
- уровень реализации бизнес-логики;
- уровень управления бизнес-процессами;
- уровень бизнес-управления (бизнес-правила, машины состояний).

Уровень сопряжения. На данном уровне работают адаптеры, которые отслеживают события, происходящие в приложениях и в интеграционной подсистеме, и трансформируют объекты, передаваемые из приложения в транспортный слой и обратно, решая проблему разных интерфейсов.

Для построения слоя сопряжения можно использовать либо готовые адаптеры, поставляемые производителем интеграционной платформы, либо адаптер может быть разработан интегратором. Возможности адаптеров оперировать сложными бизнес-объектами, уже имеющимися в интегрируемых прикладных системах; настройка интерфейсов, адресной информации и процедур трансформации, применяемых в готовых адаптерах вместо написания кода, позволяют ускорить внедрение интеграционного решения. Обычно разработчики платформ предоставляют документацию и инструментарий для разработки адаптеров.

Адаптеры, используемые в составе интеграционных платформ, можно разделить на две большие группы: технологические адаптеры и адаптеры для приложений.

Технологические адаптеры используются для сопряжения стандартных технологических компонент и используются преимущественно в тех случаях, когда интегрируемое приложение не имеет открытых API и создание прикладных адаптеров невозможно, однако остается возможность интеграции с источником данных, которые используют такое приложение, в частности с его базой данных. Примерами технологических адаптеров могут служить JDBC- и JMS-адаптеры.

Адаптеры приложений разрабатываются специально для интеграции с конкретным приложением, например для сопряжения с ERP системами, такими как SAP и PeopleSoft.

Транспортная подсистема. Транспортная подсистема (ТП), работающая в составе ESB, позволяет интегрируемым приложениям

общаться друг с другом асинхронным образом, оформляя информацию в виде сообщений и передавая ее, например через систему очередей. Асинхронность обмена сообщениями позволяет интегрировать слабосвязанные приложения, ничего не знающие друг о друге, т. е. разработанные без учета совместимости форматов и моделей данных и без учета способности реагировать немедленно на полученную информацию. На данном уровне работают модули, отвечающие за управление и безопасность информации на уровне транспорта, кроме того, может выполняться простая обработка сообщений и их маршрутизация.

ТП предоставляет разработчикам простой высокоуровневый интерфейс для подключения прикладных подсистем, что позволяет уменьшить объем низкоуровневого программирования.

Уровень реализации бизнес-логики. На данном уровне реализуются функции, связанные с трансформацией и интеллектуальной маршрутизацией сообщений, которыми обмениваются интегрируемые прикладные системы. На этом уровне работают, в частности, брокеры сообщений, которые принимают сообщения от интегрируемых систем через ТП и через нее же отправляют их после анализа и обработки системам-адресатам.

Обычно брокер выполняет следующие действия:

- принимает сообщения и направляет их по одному или нескольким адресам;
- преобразовывает форматы сообщений;
- выполняет агрегирование сообщений, разбивает поступающее сообщение на несколько сообщений и отправляет их по назначению, затем повторно собирает ответы в одно сообщение и возвращает их источнику;
- взаимодействует с внешними репозиториями с целью сохранения сообщений и получения данных;
- реализовывает вызовы Web-служб для выборки данных;
- обрабатывает сообщения о события и ошибках;
- выполняет маршрутизацию сообщений по явно указанному адресу или по содержимому, или по теме сообщения.

Брокеры используются преимущественно в интеграционной среде, основанной на передаче сообщений.

Уровень управления бизнес-процессами. На данном уровне работают системы управления бизнес-процессами, при этом для управления бизнес-процессами используется BPEL. Управление осуществляется посредством вызова Web-сервисов.

Уровень бизнес-управления. Данный уровень является надстройкой над уровнем управления бизнес-процессами. На этом уровне управление бизнес-процессами осуществляется в терминах предметной области. При этом используются два основных механизма: бизнес-правила и машины состояний, которые применяются для реализации событийного управления.

Типовая структура интеграционной системы на базе ESB. На рис. 6.11 показана обобщенная структура интеграционной системы на базе ESB второго поколения.

Ядром системы является корпоративная интеграционная шина ESB, которая позволяет интегрировать бизнес-приложения, имеющие JMS-интерфейсы и поддерживающие работу с Web-сервисами. Кроме того, ESB позволяет пользователям работать с порталом, и поддерживает взаимодействие со службами каталогов. Обычно разработчики предлагают набор как платформенных, так и технологических адаптеров. В состав платформенных адаптеров обычно входят для взаимодействия с ERP системами, такими как SAP R/3, Siebel, Oracle Applications, PeopleSoft и др., а также средства для поддержки взаимодействий B2B. В состав технологических адаптеров обычно входят адаптеры для работы с компонентами (COM, DCOM, .Net, CORBA), адаптеры для работы с данными и Java-объектами. Кроме того, интегратор может сам разрабатывать необходимые адаптеры, например для взаимодействия с унаследованными системами.

В состав практически всех ESB входят BPEL-процессоры (движки) и процессоры (движки) для работы с бизнес-правилами.

Для нормального функционирования ESB должна иметься определенная информационная инфраструктура. Часто эта инфраструктура определяется как набор сервисов, в состав которых обычно входят следующие группы сервисов:

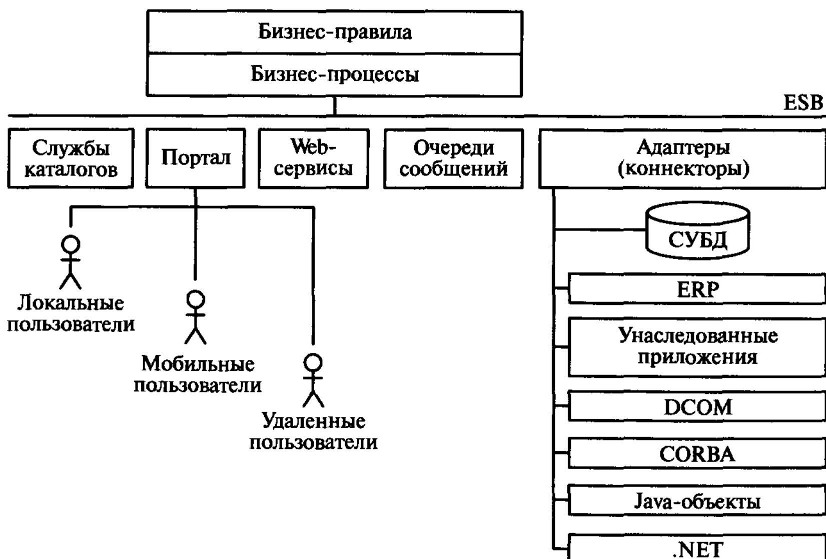


Рис. 6.11. Обобщенная структура интеграционной системы на базе ESB второго поколения

- инфраструктурные сервисы;
- сервисы управления и мониторинга;
- сервисы оптимизации;
- сервисы разработки.

Назначение и состав данных сервисов будет рассмотрен ниже на примерах конкретных систем.

Следует заметить, что структура ESB носит обобщенный характер и реальные системы могут использовать существенно отличные архитектурно-структурные решения.

6.7.3. Существующие решения ESB

Как указывалось выше, начиная приблизительно с 2005 г. на рынке ИТ-продуктов предлагается очень много разнообразных интеграционных продуктов, кроме того, имеется достаточно много бесплатных ESB. Практически все крупные компании, работающие в области КИС, предлагают собственные интеграционные решения.

Далее описан ряд альтернативных подходов к использованию интеграционных технологий. Достаточно подробно описаны подходы, развиваемые фирмами Microsoft и IBM. Подобный выбор обусловлен тем, что система BizTalk от Microsoft является наиболее распространенной, а система WebShpere от IBM — наиболее мощной, при этом BizTalk ориентирована на средний и малый бизнес, а WebShpere — прежде всего на крупный бизнес.

Microsoft BizTalk Server. BizTalk Server представляет собой семейство программных продуктов компании Microsoft, предназначенных для реализации интеграционных решений как типа A2A, так и типа и B2B. BizTalk также реализует функции сервера приложений.

BizTalk позволяет реализовывать управление бизнес-процессами на внутрикорпоративном и межкорпоративном уровнях. Используя BizTalk, организации могут создавать распределенные бизнес-процессы, интегрирующие различные приложения внутри предприятия, а также реализующие надежное и безопасное взаимодействие с партнерами организации через локальную сеть и Интернет.

BizTalk является одним из старейших приложений на рынке интеграционных решений, первая версия которого появилась еще в 2000 г. На момент написания книги были созданы следующие версии BizTalk: BizTalk Server 2000, BizTalk Server 2002, BizTalk Server 2004 (переписана с использованием .NET), BizTalk Server 2006, BizTalk Server 2006 R2, BizTalk Server 2009, BizTalk Server 2010.

На протяжении своего существования архитектура BizTalk претерпела существенные изменения. Первая версия появилась еще до широкого распространения СОА и была ориентирована преимущественно на работу с очередями сообщений и B2B-интеграцию.

Последняя версия — BizTalk Server 2010 позиционируется как ESB с развитыми функциональными возможностями.

BizTalk Server 2010, по существу, является версией BizTalk Server 2009 R2, так и не увидевшей свет в качестве коммерческого продукта. Предлагаемое решение выполняет функцию сервисной шины предприятия (enterprise service bus) и помогает реализовать поддержку сервисно-ориентированной архитектуры (SOA) в гетерогенных вычислительных средах.

Отличительная особенность развиваемого фирмой Microsoft подхода к созданию линейки продуктов BizTalk состоит в том, что разработчики делают особый акцент на интеграции BizTalk с собственными серверными продуктами. В частности, BizTalk Server 2010 ориентирована на использование с такими широко используемыми продуктами, как SharePoint Server 2010, Windows Server 2008 R2, SQL Server 2008 R2, .NET Framework 4 и Visual Studio 2010. System Center и Windows Server AppFabric.

Взаимодействие с приложениями BizTalk осуществляет через адаптеры. В дистрибутиве с BizTalk поставляются большое число как технологических адаптеров, так и адаптеров приложения, в частности, технологические адаптеры для HTTP, SOAP, FTP, POP3, SMTP, SQL, MSMQ, MQSeries, Microsoft SharePoint. Кроме того, поставляются адаптеры к таким существующим корпоративным системам, как SAP, Microsoft Dynamics CRM, Oracle eBusiness Suite, и адаптеры поддержки таких корпоративных стандартов, как EDI и RosettaNet и др. Разработчики систем поставляют многие другие адаптеры.

Обобщенная структура BizTalk Server показана на рис. 6.12.

Основу BizTalk составляет процессор (движок), в состав которого входят две подсистемы:

- подсистема работы с сообщениями (Messaging);
- подсистема управления бизнес-процессами (Orchestration).

Подсистема работы с сообщениями обеспечивает поддержку работы с разными приложениями посредством адаптеров. Данная

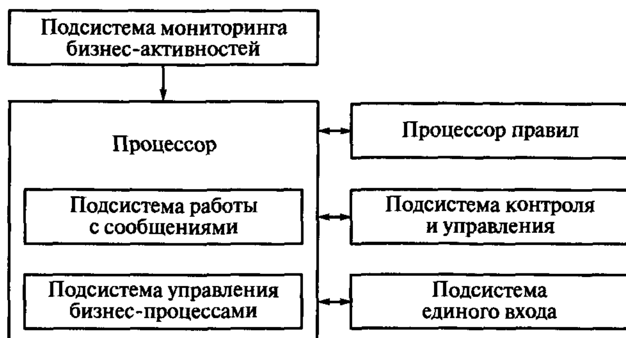


Рис. 6.12. Обобщенная структура BizTalk Server

подсистема позволяет работать с большим числом протоколов и форматов данных, включая Web-сервисы и очереди сообщений.

Подсистема управления бизнес-процессами представляет собой BPEL-движок, реализующий управление бизнес-процессами в форме оркестровки. Данный движок работает «поверх» подсистемы работы с сообщениями.

Кроме перечисленных выше, в состав BizTalk входят еще четыре компонента:

- процессор бизнес-правил (Business Rule Engine);
- подсистема контроля и управления (Group Hub);
- подсистема единого входа (Enterprise Single Sign-On facility);
- подсистема мониторинга бизнес-активностей (Business Activity Monitoring).

Процессор бизнес-правил обеспечивает поддержку работы с бизнес-правилами.

Подсистема контроля и управления обеспечивает для разработчиков и администраторов возможность мониторинга и управления движками в процессе работы.

Подсистема единого входа позволяет реализовывать обмен аутентификационной информацией между Windows и не-Windows-системами.

Подсистема мониторинга бизнес-активностей позволяет отслеживать ход выполнения бизнес-процессов в бизнес-терминах, при этом пользователи могут определять состав отображаемой информации.

BizTalk Server хорошо поддержан инструментальными средствами разработки и сопровождения, в состав которых входят:

- BizTalk Server Orchestration Designer;
- BizTalk Editor;
- BizTalk Mapper;
- BizTalk Messaging Manger;
- BizTalk Framework;
- BizTalk Administration Tool;
- Pipeline Editor.

BizTalk Server Orchestration Designer представляет собой инструментальное средство для разработки бизнес-процессов.

BizTalk Editor представляет собой редактор XML-документов.

BizTalk Mapper — это среда, в которой можно в графическом виде определить процесс преобразования XML-документов.

BizTalk Messaging Manger позволяет определить процесс обмена информацией.

BizTalk Framework позволяет описывать маршрутизацию и анализировать принимаемые и передаваемые данные.

BizTalk Administration Tool позволяет реализовывать процесс аналитической обработки о работе BizTalk.

Pipeline Editor позволяет от начала до конца отслеживать процесс обработки документа [27].

Интеграционные решения фирмы IBM. Фирма IBM предлагает интеграционные решения в рамках семейства программных продуктов WebSphere. WebSphere относится к категории middleware — промежуточного программного обеспечения, которое позволяет приложениям электронного бизнеса (e-business) работать на разных платформах.

WebSphere базируется на использовании открытых стандартов, таких как XML и Web-сервисы, которые реализуются средствами JEE. WebSphere разрабатывается в лабораториях IBM в течение многих лет, и на данный момент последней является версия 7.0, которая появилась в середине 2008 г. Эталонная СОА показана на рис. 6.13.

Идея подхода, развиваемого в рамках WebSphere, состоит в том, что предлагается универсальная модель вызова компонентов и универсальное представление данных. Универсальная модель вызова реализована в виде архитектуры сервисных компонентов (Service Component Architecture, SCA), а основой универсального представления данных служат бизнес-объекты (Business Object).

При использовании SCA все артефакты (процессы, бизнес-правила, задачи персонала (Human Tasks) и т. д.) описываются как компоненты сервисов с детально проработанными интерфейсами.

Концепция SCA дает возможность инкапсулировать бизнес-логику интеграционного решения внутри соответствующих модулей. При таком подходе изменение компонента сервиса внутри модуля не повлияет на остальные модули, входящие в состав интеграционного решения, поскольку интерфейс измененного модуля остается прежним. Все интеграционные артефакты представляются как ком-

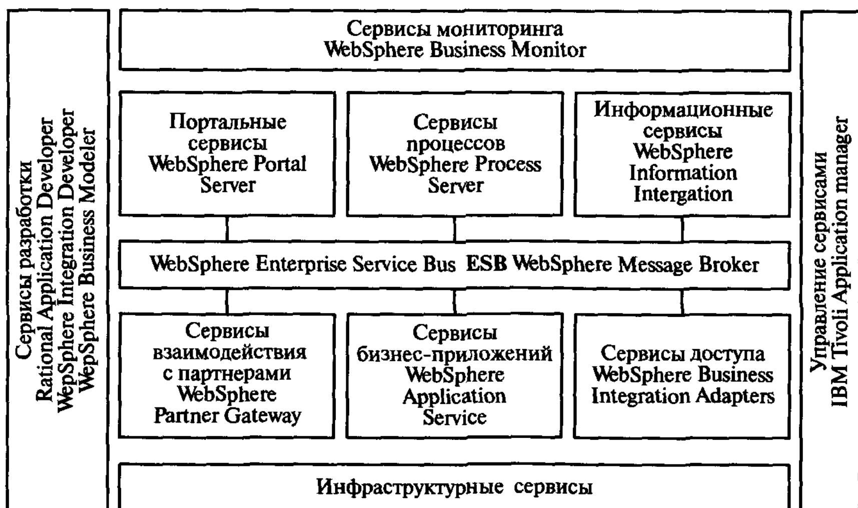


Рис. 6.13. Эталонная СОА

поненты SCA-сервисов, которые могут запускаться в асинхронном и в синхронном режимах.

Еще одним ключевым понятием данного подхода является понятие бизнес-объекта. Бизнес-объект — это расширение объекта типа Service Data Object (SDO). В отличие от объектов SDO бизнес-объекты включают несколько расширений, используемых для более детального представления данных, которыми обмениваются сервисы архитектуры SCA.

Принципиальным является то, что реализация как SCA, так и бизнес-объектов, основана на существующих стандартах. Интерфейсы любого компонента сервиса SCA описываются в терминах WSDL, а бизнес-объекты описываются с использованием XML.

Пакет продуктов IBM, входящий в состав WebSphere, для решения задач интеграции называется IBM WebSphere Business Integration и реализует такие функциональные возможности, как:

- моделирование бизнес-процессов;
- A2A-интеграцию приложений;
- B2B- и B2C-интеграцию;
- мониторинг бизнес-процессов;
- оптимизацию бизнес-процессов.

В состав данного пакета входят следующие продукты:

- IBM WebSphere Process Server — сервер бизнес-процессов;
- IBM WebSphere Enterprise Service Bus (ESB) — интеграционная шина;
- IBM WebSphere Message Broker — вариант интеграционной шины, основанный на WebSphere MQ;
- IBM WebSphere Partner Gateway — шлюз для поддержки партнерских связей;
- IBM WebSphere Business Monitor — средство мониторинга бизнес-процессов (Business Activity Monitoring);
- IBM WebSphere Business Modeler — средство анализа и моделирования бизнес-процессов;
- IBM WebSphere Integration Developer — среда разработки интеграционных сервисов и составных сервис-ориентированных приложений.

WebSphere Process Server. WebSphere Process Server реализован на базе WebSphere Application Server; имеет широкие функциональные возможности и позволяет решать практически любые интеграционные задачи и полностью соответствует принятой в IBM эталонной архитектуре интеграции (WebSphere Integration Reference Architecture).

В Process Server все ключевые сервисы реализованы на основе собственного провайдера JMS, обеспечивающего полную совместимость с существующими решениями на базе WebSphere MQ, поддержку Web-сервисов и архитектуры Java 2 Connector Architecture.

Архитектура Process Server приведена на рис. 6.14. Process Server работает поверх сервера приложений, в качестве которого выступа-

Рис. 6.14. Архитектура Process Server

ет WebSphere Application Server, и имеет три уровня: уровень ядра СОА, уровень обеспечивающих сервисов и уровень сервисных компонентов.

Общая инфраструктура событий (Common Event Infrastructure, CEI) обеспечивает фиксацию событий, которые можно использовать для мониторинга приложений средствами Business Monitor или других продуктов IBM.

К обеспечивающим сервисам относятся компоненты, необходимые для любого интеграционного решения. К этой группе сервисов относятся:

- сервисы отображения интерфейсов;
- сервисы отображения бизнес-объектов;
- сервис-реализации отношений;
- селекторы.

Сервисы отображения интерфейсов: во многих случаях интерфейсы существующих компонентов имеют различный синтаксис, но семантически идентичны. Рассматриваемые сервисы позволяют вызывать такие компоненты путем трансляции соответствующих обращений.

Сервисы отображения бизнес-объектов используются для преобразования бизнес-объекта из одного типа в другой.

Сервис реализации отношений: сценарии бизнес-интеграции часто предполагают обращение к одним и тем же данным, используемым несколькими серверными системами, например, ERP и CRM. Типовая проблема при синхронизации бизнес-объектов состоит в том, что разные серверные системы используют разные ключи для представления одного и того же объекта. Сервис отношений служит для установления соответствия между идентичными объектами в разных серверных системах.

Селекторы — это компоненты, поддерживающие механизм динамического выбора и вызов различных сервисов, использующих один и тот же интерфейс.

На уровне сервисных компонентов Process Server поддерживает работу со следующими типами компонентов:

- бизнес-процессы;
- бизнес-правила;
- машины бизнес-состояний;
- задачи персонала.

Управление *бизнес-процессами* реализуется в соответствии со стандартом WS-BPEL. Бизнес-процессы можно создавать с помощью инструмента WebSphere Integration Developer или импортировать из бизнес-модели, созданной с помощью инструмента WebSphere Business Modeler.

Бизнес-правила — это средство внедрения и отслеживания бизнес-политики посредством вывода бизнес-функции. Данный механизм обеспечивает динамическое внесение изменений в бизнес-процесс для повышения оперативности бизнес-среды. Для разработки бизнес-правил существует инструмент на базе платформы Eclipse. В состав WebSphere Process Server входит также работающий в реальном времени Web-инструмент бизнес-аналитики, по мере необходимости обновляющий бизнес-правила без нарушения работы других сервисов.

Машины бизнес-состояний (*Business State Machine*) — один из способов моделирования бизнес-процесса. Этот способ позволяет организации описать свои бизнес-процессы с помощью состояний и событий, что в некоторых случаях оказывается проще, чем моделировать бизнес-процессы с помощью графов.

Задачи персонала (Human Tasks) в решении WebSphere Process Server — это автономные компоненты, с помощью которых сотрудникам поручается определенная работа или вызываются какие-либо другие сервисы. Кроме того, инструмент Human Task Manager позволяет описывать специальные задания и отслеживать их выполнение. Для получения информации о персонале предназначены существующие каталоги LDAP.

В состав WebSphere Process Server входит расширяемый Web-клиент, который может служить для работы с заданиями и процессами.

Все функции WebSphere Process Server настраиваются и администрируются с помощью специальных расширений консоли администратора WebSphere Application Server и различных конфигурационных средств.

WebSphere Process Server использует все возможности сервера приложений при управлении транзакциями, безопасностью, кластеризацией и рабочей нагрузкой, формируя среду бизнес-интеграции с высокой степенью масштабируемости и надежности. Он обеспечивает также полную поддержку транзакций ACID при реализации бизнес-процессов, как коротких (одна транзакция от начала до конца), так и длительных (множество транзакций). В состав продукта

входят инструменты восстановления Recovery Manager и Recovery Console. Если в процессе выполнения какого-либо приложения происходит ошибка, Process Server позволяет администратору через консоль Recovery Console применить соответствующие процедуры к отказавшему приложению.

Решение WebSphere Process Server дает множество возможностей для интеграции. Помимо импорта/экспорта собственных объектов (SCA-компонентов, Web-сервисов, JMS и Enterprise Java Session Beans) это решение обеспечивает совместимость с существующими приложениями на базе WebSphere MQ.

В состав Process Server входит также инструмент WebSphere Integration Developer.

В состав WebSphere Integration Developer входит инструмент Assembly Editor, с помощью которого разработчик объединяет различные компоненты в модуль и указывает, при помощи каких сервисных интерфейсов этот модуль будет взаимодействовать с внешними потребителями. В качестве сервисов могут выступать импортируемые компоненты, например, Java Beans и Web-сервисы, а также компоненты сервисов, предоставляемые WebSphere Process Server.

WebSphere Enterprise Service Bus (*ESB*). Гибкая коммуникационная инфраструктура для интеграции приложений и сервисов позволяет уменьшить число и сложность интерфейсов между компонентами системы. ESB распределяет сообщения между сервисами, конвертирует транспортные протоколы и форматы сообщений между источником запроса и сервисом, а также управляет бизнес-событиями различных источников.

ESB позволяет организациям сосредоточиться на основных задачах бизнеса, а не на ИТ-инфраструктуре, необходимой для связывания программ между собой, и добавлять новые сервисы или изменять существующие при минимальном воздействии на работу уже имеющихся сервисов. Он обеспечивает перемещение данных между приложениями, распознавание формата данных, интеллектуальную маршрутизацию и преобразование форматов данных XML. С его помощью можно использовать WebSphere MQ в качестве инфраструктуры для коммуникационного взаимодействия между приложениями, а также применять другие коммуникационные протоколы, такие как JMS или HTTP.

WebSphere ESB построен на основе WebSphere Application Server. Продукт можно использовать совместно с WebSphere Integration Developer или с Rational Application Developer (если нужно писать Java-код). Для построения единой сервисной шины предприятия IBM (в дополнение к ESB) предлагает решение IBM WebSphere Message Broker.

WebSphere Message Broker. Этот брокер расширяет функции обмена сообщениями WebSphere MQ, добавляя к ним средства маршрутизации, преобразования и возможность работы в режиме публи-

кации/подписки сообщений. Message Broker может выполнять разнообразные фукнции, в числе которых следующие:

- маршрутизация сообщений для нескольких целевых мест назначения на основе содержания сообщения или его заголовка;
- преобразование сообщений в разные форматы;
- дополнение содержимого сообщений в процессе передачи (например, за счет поиска в базе данных, выполняемого брокером сообщений);
- сохранение информации, извлеченной из сообщений в процессе их передачи, в базу данных;
- публикация сообщений и использование подписчиками критериев, основанной на теме или содержании.

WebSphere Message Broker предоставляет опции масштабируемости в виде копий потоков сообщений и групп выполнения и упрощает интеграцию существующих приложений и Web-сервисов за счет преобразования и маршрутизации сообщений SOAP. Кроме того, он может выступать как промежуточное звено между Web-сервисами и другими моделями интеграции и обеспечивать транспортировку данных WebSphere MQ для корпоративной, мобильной и широковещательной связи, а также для передачи данных в режиме реального времени.

В качестве инструментальных средств для работы с данным продуктом используется Message Broker Toolkit for WebSphere Studio, который реализован на базе WebSphere Message Broker. Он может работать на разных платформах и поддерживать обширный список ОС (Windows Server, AIX, Solaris, HP-UX, Linux для Intel, Linux для zSeries и z/OS), в качестве репозитария для хранения информации о конфигурации используется база данных DB2.

IBM WebSphere MQ. WebSphere MQ — это промежуточное ПО IBM, предназначенное для работы с очередями сообщений, которой поддерживает более 30 платформ и реализует следующие основные функции:

- обеспечивать гарантированную однократную доставку сообщений;
- поддерживать не зависящую от времени коммуникацию;
- обеспечивает высокую пропускную способность, которая может составлять более 250 млн сообщений в день;
- поддерживать безопасные взаимодействия через защищенные каналы.

WebSphere Partner Gateway. Partner Gateway предназначен для организации взаимодействий типа B2B.

Решение интегрирует внешние процессы и партнерские сообщества с внутренними процессами и инфраструктурами компании, предоставляя широкие возможности управления профилями партнеров и простые надежные и защищенные средства для обмена сообщениями на уровне B2B.

Partner Gateway формирует единую среду управления B2B-взаимодействиями с партнерами, поддерживающую такие форматы данных и протоколы передачи сообщений на основе EDI и XML для B2B-интеграции, как AS1, AS2 и RosettaNet.

WebSphere Business Monitor. Продукт отображает поступающие в режиме реального времени данные о событиях, происходящих в системе в терминах работы с сообщениями, и реализует функции, связанные с уведомлением и предупреждением ключевых пользователей о происходящих событиях.

В состав продукта входят два основных компонента: пульт управления рабочими процессами и пульт управления бизнесом.

Пульты управления реализованы в виде страниц WebSphere Portal с картами сравнительных показателей и поддерживают многомерный анализ и могут формировать отчетность.

Пользователь может выполнять фильтрацию отчетов, а компонент Adaptive Action Manager инициирует выбранные действия или последовательности действий в реальном времени на основании установленных правил и политик.

WebSphere Business Modeler. Продукт реализован на базе Eclipse и содержит программные средства, позволяющие бизнес-аналитикам графически моделировать бизнес-процессы. В ходе моделирования система основана на использовании шаблонов. Шаблон представляет собой набор операций, необходимых для достижения какой-либо бизнес-цели. Формализация и последующее применение шаблонов позволяет оптимизировать и автоматизировать соответствующие бизнес-процессы.

Данный инструмент может моделировать бизнес-процессы, элементы бизнеса (документы и продукты), ресурсы (персонал и физические активы) и структурные отношения между элементами организации. При работе в среде WebSphere Business Modeler можно выбрать соответствующий профиль пользователя, который задает требуемый уровень детализации при работе с моделями. Базовый профиль (Basic Business Modeling) предназначен для бизнес-аналитиков, работающих с высокоуровневым представлением модели бизнес-процесса. Промежуточный профиль (Intermediate Business Modeling) предполагает более подробное описание технических деталей модели, включая бизнес-правила и логику бизнеса, которые применяются к элементам модели. Профиль высшего уровня (Advanced Business Modeling) нацелен на разработку детальных моделей бизнес-процессов, на основании которых создаются программные приложения.

WebSphere Integration Developer. В WebSphere Integration Developer входят средства тестирования, отладки и развертывания для разработки решений. Многократное использование компонентов упрощается за счет онлайновых модулей и библиотек. В его составе также имеются готовые конструкции для построения динамических процессов, включая бизнес-правила, машины бизнес-состояний и селекторы,

мероприятия и ролевые функции для задач. Продукт устанавливается на WebSphere Process Server.

Сочетание WebSphere Integration Developer с другими инструментами IBM, с одной стороны, и WebSphere Process Server и Application Server, с другой, образует платформу для проектов бизнес-интеграции и служит хорошим решением для реализации гибких сред на базе СОА. При необходимости эта среда может интегрироваться с инструментом бизнес-аналитики IBM WebSphere Business Modeler и с инструментом разработки JEE- и Web-решений IBM Rational Application Developer.

Выбор продукта для реализации ESB определяется требованиями конкретной ситуации. В составе IBM WebSphere Business Integration имеется три продукта, обладающих функциональностью ESB:

- IBM WebSphere Process Server;
- WebSphere Enterprise Service Bus;
- WebSphere Message Broker.

WebSphere Enterprise Service Bus предназначен для обеспечения базовых возможностей ESB, в первую очередь для среды, основанной на Web-службах. Этот продукт основывается на WebSphere Application Server, который является базой транспортного уровня. В WebSphere Enterprise Service Bus к этой основе добавляется посреднический уровень, основанный на программной модели SCA, который обеспечивает интеллектуальные возможности связи. Если у заказчика в среде используется много Web-служб, то скорее всего WebSphere Enterprise Service Bus будет наилучшим решением.

WebSphere Message Broker представляет собой более совершенное ESB-решение с дополнительными возможностями интеграции, такими как универсальная связь и трансформации «из любого в любой формат» для систем, ориентированных на работу с данными. Этот продукт может осуществлять интеграцию служб, а также интеграцию с приложениями, не использующими службы. Как правило, заказчикам, которым требуется высокопроизводительный продукт в среде, ориентированной на работу с сообщениями, следует использовать WebSphere Message Broker.

IBM WebSphere Process Server целесообразно использовать в случае, если требуется работать со сложными бизнес-процессами, используя, в частности механизмы работы с бизнес-правилами.

Кроме всего прочего следует отметить, что перечисленные выше продукты существенно различаются по стоимости.

Эти продукты также можно использовать в сочетании. Существует два основных сценария, в которых возможно применение такого сочетания:

- в первом случае оба ESB-продукта соединяются друг с другом для создания в масштабе предприятия ESB, объединяющей в себе возможности поддержки Web-служб и интеграцию приложений для обмена сообщениями;

• во втором случае WebSphere Message Broker играет роль центральной ESB, а WebSphere Enterprise Service Bus обеспечивает обработку сообщений, отправляемых в филиалы, хранилища, базы данных и т. п. [43].

Другие коммерческие и свободно распространяемые системы интеграции КИС. Перечень и краткая характеристика отдельных как коммерческих, так и свободно распространяемых систем интеграции КИС класса ESB приведены ниже. Следует иметь в виду, что данный список далеко не полный.

BEA AquaLogic (*BEA Systems*) [34]. Представляет собой семейство продуктов, охватывающее полный жизненный цикл сервисно-ориентированной архитектуры (SOA), с помощью которых можно создавать, развертывать и управлять сервисами. BEA AquaLogic позволяет изолировать сложные и разнообразные технологии и программные среды, применяемые на предприятии, и создать нейтральный контейнер для функционирования бизнеса. Данный пакет дает интегратору полный набор инструментов для внедрения и управления сервисной архитектурой организации.

Apache ServiceMix (*The Apache Software Foundation*) [26]. Представляет собой Java-фреймворк, ориентированный на СОА и ESB и поддерживающий спецификации API Java Business Integration (JBI) и OSGi. Этот фреймворк предназначен для реализации главной идеи JBI — построение решений на основе сервисов по компонентному принципу. Как и многие другие проекты Apache, ServiceMix имеет интеграцию с таким актуальным для Java-программистов решением, как Spring. ServiceMix включает в себя полностью реализованный JBI-контейнер, реализующий все спецификации JBI. ServiceMix предоставляет простое клиентское API для работы с JBI-компонентами и сервисами, а также реализацию WS Notification. ServiceMix может использоваться совместно с Apache Camel, ActiveMQ и Apache Synapse.

Apache Camel (*The Apache Software Foundation*) [41]. Представляет собой открытый кроссплатформенный Java-фреймворк, который позволяет интегрировать приложения в простой и понятной форме. Основу составляет Apache Camel — процессор маршрутизации (routing engine). Он позволяет определять собственные правила маршрутизации, источники сообщений, методы обработки сообщений и их получателей. Camel вводит простой интеграционный язык, с помощью которого можно определить сложные правила маршрутизации, подобно описанию бизнес-процессов. Один из фундаментальных принципов Camel в том, что он не навязывает никакой канонической модели данных. Сообщения могут быть в любом виде, а не только в виде XML. Camel безразличен к формату обрабатываемых данных. Что приходит извне, то и попадает на обработку в тот или иной маршрут. В то же время Camel поддерживает высокий уровень абстракции, что позволяет обмениваться данными с разными системами без оглядки на специфику протокола. Число реализован-

ных протоколов, подключаемых к ядру в виде компонентов, составляет уже больше 70. Такой выбор архитектуры позволяет преобразовывать сообщения только там, где это действительно нужно. Все это делает Camel гибким и быстрым. Он легко встраивается в проекты, где нужна мощнейшая поддержка маршрутизации сообщений.

Mule (*MuleSoft*) [35]. Инструментарий, разработанный в качестве альтернативы громоздким платформам интеграции, для использования которых необходимы специальные навыки и кропотливая работа. В основе Mule лежит идея упростить труд интегратора, дав ему возможность сосредоточиться на создании необходимой бизнес-логики. Инструментарий можно применять как в простейших проектах — для связи двух конечных точек, так и в качестве ESB.

Open ESB (*Oracle*) [47]. Является проектом Sun Microsystems и ведется как один из проектов Java.net. Так же, как и ServiceMix, эта система представляет собой реализацию спецификаций JBI. По функционалу Open ESB сходен с ServiceMix. Главное отличие в том, что Open ESB ориентирован на Glassfish Application Server.

6.8. Сервисно-ориентированная архитектура и сервисно-ориентированная организация

Когда говорят об уровнях зрелости применительно к ИТ-сфере, то чаще всего имеют в виду Capability Maturity Model Integration (CMMI) — набор моделей (методологий) совершенствования процессов в организациях разных размеров и видов деятельности, которая содержит набор рекомендаций в виде практик, реализация которых позволяет достичь целей, необходимых для успешных определенных видов деятельности.

Набор моделей CMMI включает три модели: CMMI for Development (CMMI-DEV), CMMI for Services (CMMI-SVC) и CMMI for Acquisition (CMMI-ACQ). Наиболее известной является модель CMMI for Development, ориентированная на организации, занимающиеся разработкой программного и аппаратного обеспечения, а также комплексных систем. Все действующие версии моделей имеют номер 1.3 (вышли в ноябре 2010 г.).

CMMI является развитием методологии CMM, которая разрабатывалась со второй половины 1980-х гг. Software Engineering Institute (SEI) в университете Карнеги-Меллона (Carnegie Mellon University). [32].

CMMI определяет 22 процессные области (process areas). Для каждой из процессных областей существует ряд целей (goals), которые должны быть достигнуты при внедрении CMMI в данной процессной области. Некоторые цели являются уникальными, они называются специфическими (specific). Общие (generic) цели применяются к нескольким процессным областям. Цели достигаются при помощи реализации практик, которые делятся на специфические и общие. Существуют

Уровень 5	Оптимизирующий
Уровень 4	Управляемый
Уровень 3	Определенный
Уровень 2	Повторяемый
Уровень 1	Начальный

Рис. 6.15. Уровни зрелости организации

два представления CMMI: непрерывное (continuous) и ступенчатое (staged). При реализации практик CMMI с использованием непрерывного представления выбор процессных областей не фиксирован. Для оценки уровня институционализации процессной области используется шкала уровней способности (capability level) от 0 до 5 (шесть уровней). Ступенчатое представление определяет пять (1—5) уровней зрелости (maturity level) организации (рис. 6.15). Для достижения каждого уровня зрелости (кроме 1-го) необходимо выполнить требования по внедрению практик определенного набора процессных областей для достижения соответствующих целей. Уровень 1 зрелости в модели не определен.

Модель CMMI была хорошо принята разработчиками ПО и достаточно успешно применялась на практике, поэтому появилась идея распространить идеи модели, основанные на понятии зрелости на смежные области. В частности, были предложены ряд моделей зрелости СОА и модель зрелости Maturity Model for Service Oriented Enterprises. В отличие от CMMI, которые являются фактически отраслевыми стандартами, модели зрелости СОА и SOE — это предложения отдельных компаний. Хотя эти модели не столь подробно проработаны как CMMI, они достаточно полезны как для системных архитекторов, так и интеграторов [46].

Ниже приводится модель зрелости СОА, предложенная в 2005 г. фирмами Sonic Software Corporation, AmberPoint Inc., BearingPoint, Inc., Systinet Corporation 2005.

В рамках данной модели выделяются 5 уровней (рис. 6.16).

Уровень 1. Эпизодическое применение СОА (Initial Services).

Уровень 2. Архитектурный уровень (Architected Services).

Уровень 3. Бизнес-сервисы и службы взаимодействия (Business Services And Collaborative Services).

Уровень 5	Оптимизация бизнес-процессов
Уровень 4	Снятие метрик бизнес-сервисов
Уровень 3	Бизнес-сервисы и службы взаимодействия
Уровень 2	Архитектурный уровень
Уровень 1	Эпизодическое применение СОА

Рис. 6.16. Модель зрелости СОА

Уровень 4. Снятие метрик бизнес-сервисов (Measured Business Services).

Уровень 5. Оптимизация бизнес-процессов (Optimized Business Services).

Уровень 1. Эпизодическое применение СОА (Initial Services). Это низший уровень зрелости, на котором принимаются решения о целесообразности ориентации организации на использования СОА и выполняются отдельные проекты, ориентированные на реализацию СОА.

На этом уровне идет процесс внедрения основных стандартов СОА от W3, таких как XML (для определения форматов сообщений), WSDL (для описания интерфейсов) и SOAP (для вызова сервисов). Основные задачи, решаемыми на первом уровне, — ознакомление с возможностями СОА и накопление опыта работы с СОА в процессе решения конкретных задач.

На данном уровне зрелости в организации начинают внедряться ESB, реестры, поддерживающий стандарт UDDI.

В качестве примера КИС, соответствующей первому уровню зрелости, рассмотрим систему, структура которой показана на рис. 6.17 и представляет собой КИС коммерческой организации, основу которой составляет пакетное решение на базе ERP-системы, в состав которой, в частности, входит подсистема расчета заработной платы. Допустим, у менеджеров компании появилась идея внедрить систему анализа и предсказания уровня заработных плат, которая реализуется в виде отдельного сервиса. Связывание подсистемы расчета заработной платы со вновь создаваемой системой предсказания заработных плат может реализовываться разными способами, в частности, может потребоваться разработка отдельного адаптера. Доступ к обоим приложениям может быть организован через портал.

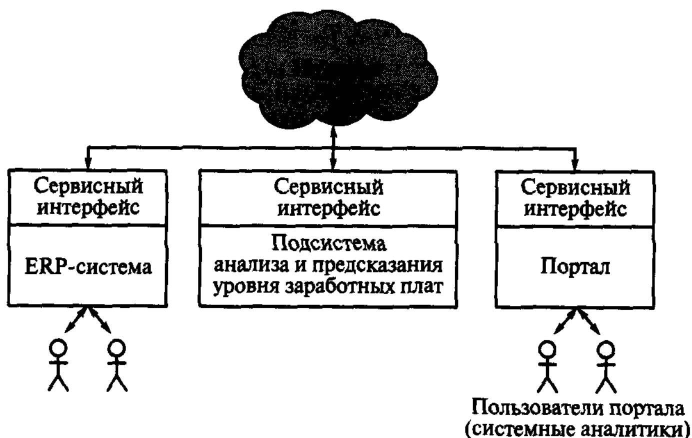


Рис. 6.17. КИС коммерческой организации, соответствующая уровню 1

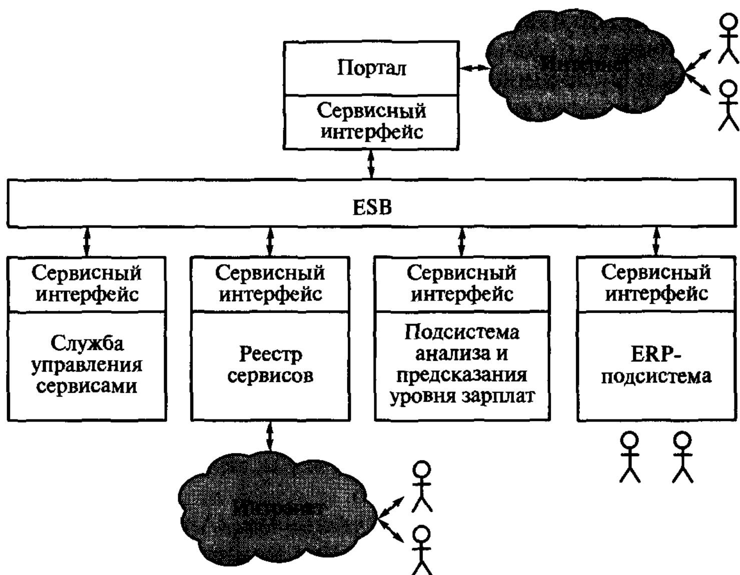


Рис. 6.18. Модернизированная КИС коммерческой организации, соответствующая уровню 1

После того как появляется определенный опыт интеграции приложений и некоторая инфраструктура, может быть принято решение о дальнейшем развитии СОА. На рис. 6.18 показана структура той же КИС, в которую включены три дополнительных элемента: ESB, Служба управления сервисами и Реестр сервисов.

Корпоративная сервисная шина (An Enterprise Service Bus — ESB) обеспечивает стандартное взаимодействие между компонентами СОА, включая Web-сервисы, реляционные базы данных, очереди сообщений и унаследованные системы. С помощью ESB легко можно интегрировать платформы, например, .NET, JEE, пакетные системы класса CRM или ERP.

Служба управления сервисами (A Service Level Management Service) представляет собой сервис, который обеспечивает управление сервисами, снятие метрик и мониторинг служб.

Реестр сервисов (A Services Registry), поддерживающий стандарт UDDI, обеспечивает централизованное хранение описаний доступных сервисов.

В состав системы данного уровня могут быть включены системы (подсистемы) ERP, анализа и предсказания уровня заработных плат.

Теперь КИС можно рассматривать как имеющую СОА архитектуру. Однако она продолжает соответствовать уровню 1, поскольку отсутствует документированная архитектура.

Уровень 2. Архитектурный уровень (Architected Services). Уровень 2 отличается от начального переходом от случайной интеграции к плановой и внедрению стандартов, в частности W3C, OASIS и WS-I стандартов. При этом предполагается, что в организации имеются документированная архитектура и план ее развития, а также политики повторного использования компонентов и политики безопасности.

Переход от уровня 1 к уровню 2 означает прежде всего переход от решения отдельных задач интеграции приложений по мере их появления к видению текущей архитектуры ИС организации и желаемой архитектуры.

Пример описанной выше КИС, соответствующий уровню 2, показан на рис. 6.19. Системы включает все ключевые элементы, которые имеет система по рис. 6.19, при этом появляются некоторые новые элементы.

Репозитарий сервисов и политик (A Services and Policies Repository) является расширением Services Registry и включает расширенное описание функциональности служб.

Сервис управления особыми ситуациями (An Exception Management Service) обеспечивает реализацию механизмов обнаружения, диагностики и реагирования на особые ситуации как на системном уровне, так и на уровне приложений.

Сервис преобразования (Message Transformation) предназначен для обеспечения интеграции служб на уровне структуры сообщений. Обыч-

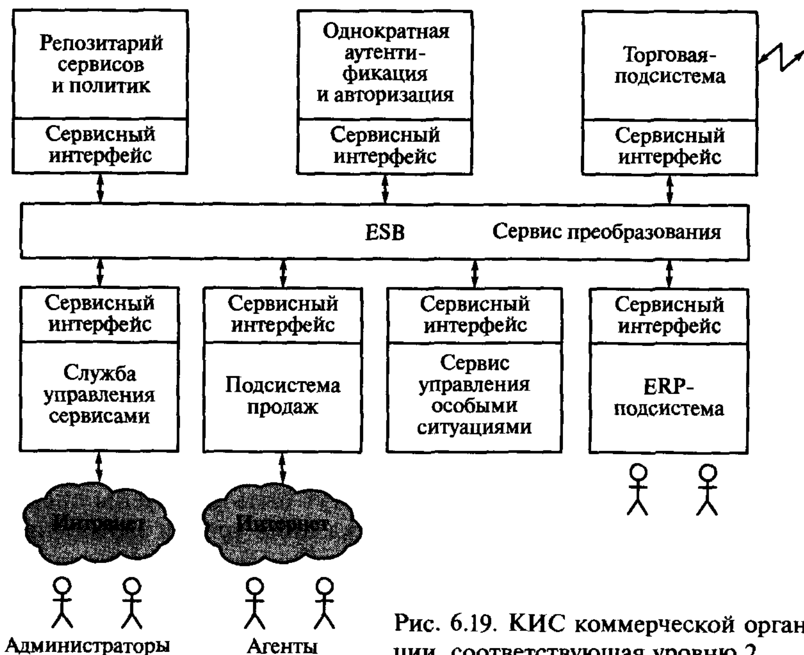


Рис. 6.19. КИС коммерческой организации, соответствующая уровню 2

но это реализуется с помощью XLST-преобразований, которые применяются к XML-сообщениям. Это происходит под управлением ESB.

Однократная аутентификация и авторизация (A Single Sign-On Service) поддерживает систему аутентификации и авторизации в рамках организации. Такие сервисы реализуются на основе стандарта OASIS SAML, который определяет форматы обмена информацией, относящейся к процедурам аутентификации и авторизации.

Кроме того, в состав КИС рассматриваемого уровня зрелости могут входить такие системы (подсистемы), как торговая, продаж, ERP.

Данная система может быть отнесена к уровню 2 зрелости, причем основным доводом в пользу отнесения ее к уровню 2 является не число реализованных сервисов, а наличие документированной архитектуры.

Уровень 3 — бизнес-сервисы н службы взаимодействия (Business Services And Collaborative Services). Уровень 3 характеризуется тем, что начинают использоваться системы управления бизнес-процессами, в которых задействованы как внутренние сервисы, так и сервисы организаций-партнеров. Кроме того, начинают использоваться бизнес-процессы с большим временем жизни и бизнес процессы, управляемые событиями, активно используется стандарт WSBPEL.

Таким образом, основной отличительной особенностью данного уровня следует считать переход от Web-сервисов к бизнес-сервисам, которые используются как внутри организации, так и для реализации партнерских связей.

Уровень 3 определяет две новые категории: бизнес-сервисы (Business Services) и сервисы сотрудничества (Collaborative Services). Иногда данный уровень разбивают на два подуровня: уровень бизнес-сервисов (3а) и уровень сервисов сотрудничества (3b).

Бизнес-сервисы ориентированы на совершенствование внутренних бизнес-процессов организации. Процессы сотрудничества ориентированы на совершенствование процессов взаимодействия с внешними партнерами.

Для того чтобы система принадлежала уровню 3, не обязательно, чтобы она поддерживала и бизнес-сервисы, и сервисы сотрудничества. Будет достаточно, чтобы поддерживался любой из видов взаимодействия.

Пример структуры рассмотренной выше КИС, соответствующей уровню 3, показан на рис. 6.20. Помимо имевшихся ранее подсистем, появляются две новые подсистемы:

• *BPEL-процессор* — подсистема управления бизнес-процессами (Business Process Management — BPM) построенная на базе BPEL-движка;

• подсистема поддержки сервисов сотрудничества (Collaboration Services), использующая протоколы взаимодействия B2B RosettaNet, которые включают стандарты XML для обмена данными при вы-

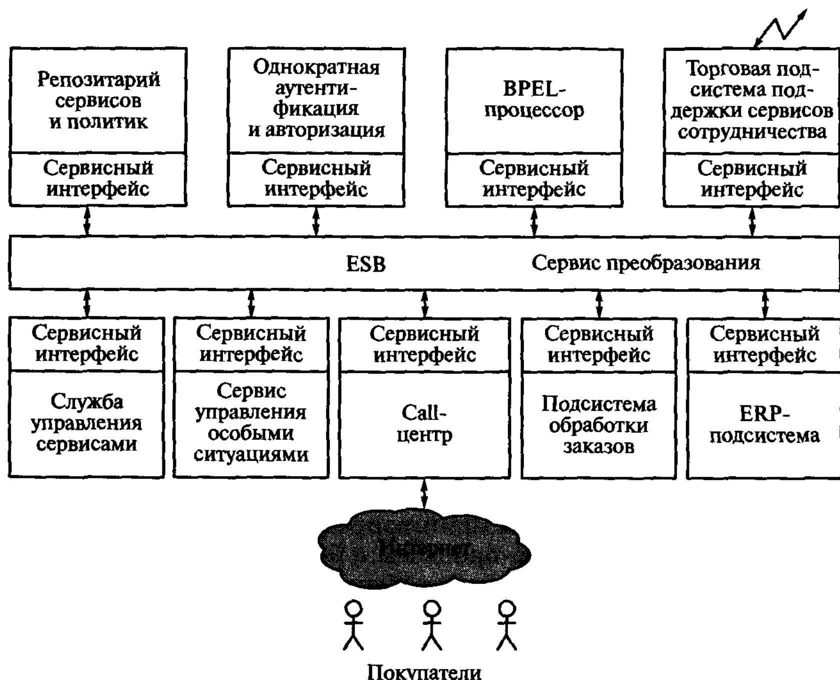


Рис. 6.20. КИС коммерческой организации, соответствующая уровню 3

полнении таких операций, как получение информации о продукте, заказы т.д.

В состав системы данного уровня зрелости могут входить также системы (подсистемы) обработки заказов и Call-центр.

Основное преимущество СОА — это возможность относительно легко модифицировать структуру бизнес-процессов за счет модификации самих сервисов. В данном примере для согласования деятельности служб продаж (Trading Service) и службы заказов (Order Management Service) используется отдельная служба согласования (Compliance Service). Сервисы, службы продаж и службы заказов остаются неизменными. Сервисами заказа и продаж пользуются Call Center Service и Online Service.

Другая характерная черта уровня 3 — использование сервисов сотрудничества (Collaborative Services), которые ориентированы на поддержание связей с внешними партнерами.

Уровень 4 — измеряемые бизнес-сервисы (Measured Business Services). Специфика этого уровня — это снятие метрик бизнес-процессов, обработка и представление результатов в терминах предметной области таким образом, чтобы обеспечить постоянную обратную связь для настройки производительности и повышения эффективности бизнес-процессов.

Однако решение об изменении структуры организации, изменении структуры бизнес-процессов принимают люди. Обычно это делают менеджеры с учетом рекомендаций, подготовленных бизнес-аналитиками.

Основной отличительной особенностью данного уровня является наличие механизма снятия и обработки бизнес-информации о функционировании системы. Полученная информация обрабатывается соответствующими подсистемами и предоставляется менеджерам.

Обработка обычно представляет собой процесс извлечения знаний из данных — data mining. На данном уровне появляются ряд новых подсистем, представленных соответствующими сервисами, основными из которых являются следующие:

- подсистема мониторинга бизнес-активности (Business Activity Monitoring, BAM), который обеспечивает обратную связь, причем результаты мониторинга накапливаются в базе данных; мониторинг осуществляется преимущественно на уровне бизнес-процессов;
- подсистема работы с бизнес-правилами (*Rule Engine*);
- подсистема работы с событиями;
- подсистема обработки событий.

Пример КИС, соответствующей уровню 4, которая является развитием рассматриваемой выше системы, показан на рис. 6.21. Система имеет ряд новых элементов. В частности она интегрирована

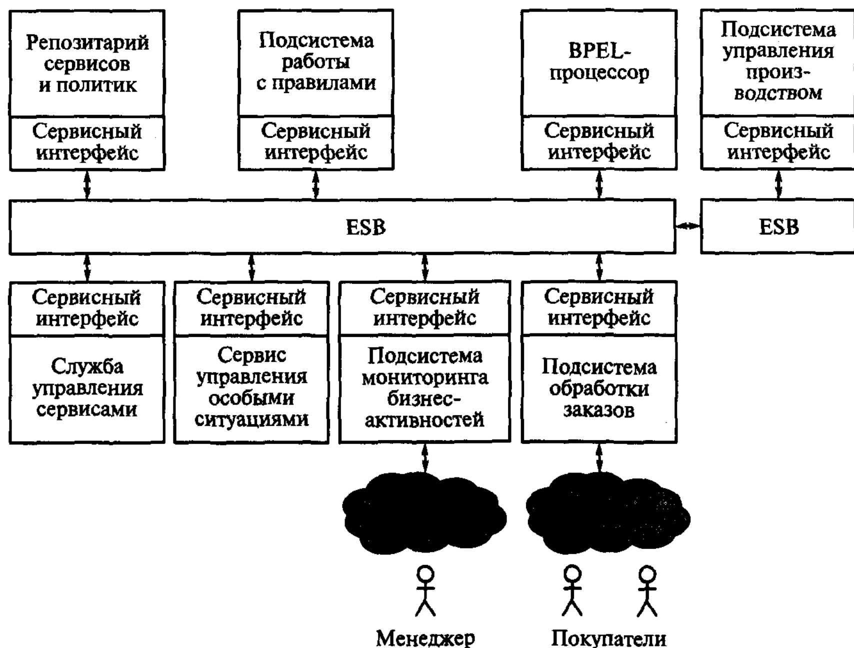


Рис. 6.21. КИС коммерческой организации, соответствующая уровню 4

с производственной подсистемой, построенной на основе отдельной ESB, включает в себя подсистемы мониторинга бизнес-активностей (PMB), (BAM), работы с правилами управления производством.

Подсистема мониторинга, бизнес-активностей реализует процесс накопления информации о событиях в БД, фильтрует информацию о событиях фильтрации и выделяет значимые события на основе правил.

В качестве пользователей системы обработки событий выступают бизнес-аналитики, ответственные за аналитическую обработку информации, и менеджеры, к которым поступает информация, уже обработанная бизнес-аналитиками. Система обработки событий работает в PMB.

Решения об изменении параметров бизнес-процессов продолжают приниматься человеком.

Уровень 5 — оптимизация бизнес-процессов (Optimized Business Services). Отличительной особенностью КИС, соответствующей уровеню 5, является наличие автоматической реакции на события. Бизнес-процессы могут трансформироваться (настраиваться). Возможно использования бизнес-интеллекта в PMB. Деятельность организации отимизируется в терминах бизнес-правил и бизнес-целей.

Структура системы, отвечающей уровню 5, показана на рис. 6.22. В качестве примера рассматривается та же система, что и была рас-

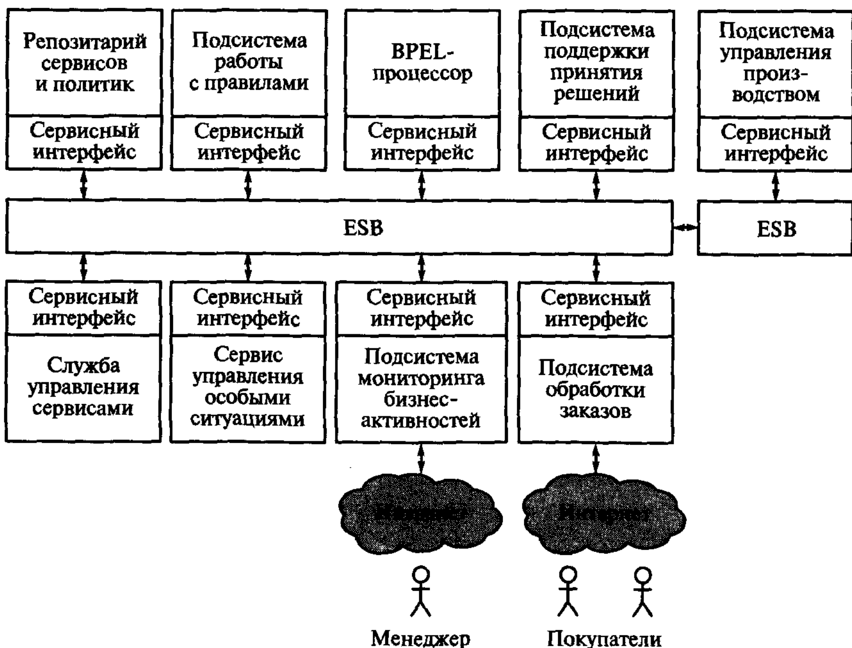


Рис. 6.22. КИС коммерческой организации, соответствующая уровню 5

смотрена в предыдущих примерах. В системе появляется новая подсистема поддержки принятия решений. Этот сервис реализует два действия:

- определение бизнес-политики и бизнес-правил;
- изменение правил в соответствии с результатами мониторинга бизнес-активностей.

Данная система может работать как в автоматическом, так и в полуавтоматическом (режиме выдачи рекомендаций) режиме.

Можно привести следующий пример использования системы, соответствующей уроню 5.

Допустим, что организация занимается выпуском и продажей компьютеров. В некоторый момент времени появляется проблема с поставками дисковых накопителей большой емкости. В этом случае система формирует бизнес-политики, направленные на повышение спроса на младшие модели посредством коррекции ценовой политики. Эти коррекции реализуются через коррекцию бизнес-правил и могут включать такие меры, как уменьшение цен, скидки, акции по распродаже и т.д.

Кроме того, достаточно просто можно организовать формирование цен в зависимости от цен на комплектующие, материалы и других факторов.

Реализовать данный подход при использовании СОА значительно легче, чем при использовании других архитектурных решений.

Рассмотренная выше модель зрелости СОА включает как технологические, так и бизнес-аспекты. На уровнях 1—3 уровень зрелости оценивается в технологических терминах, т.е. принадлежность к соответствующему уровню зрелости рассматривается с точки зрения использования тех или иных технологий. Например, наличие BPEL-процессора является необходимым, для того чтобы система могла быть отнесена к уровню зрелости 3а. Следует заметить, что применительно к уровням 4—5 используются уже термины бизнес-логики.

Модель зрелости сервисно-ориентированной организации (Maturity Model for Service Oriented Enterprises) была предложена фирмами IBM and BEA Systems приблизительно в одно время с рассмотренной выше моделью зрелости СОА и имеет целью определить модель зрелости в бизнес-терминах. Она определяется в таких терминах, как тематика выполняемых ИТ-проектов и возврат инвестиций.

Данная модель также определяет пять уровней зрелости.

Уровень 1 — базовый (SOE foundation). Это начальный уровень, на котором должны быть определены все основные бизнес-процессы и должна быть создана сервисно-ориентированная инфраструктура, определены основные показатели эффективности как сервисно-ориентированной инфраструктуры, так и бизнес-процессов.

На данном уровне зрелости предполагаются активные финансовые вложения в создание сервисно-ориентированной инфраструктуры.

Выполняемые ИТ-проекты ориентированы на создание развитой сервисно-ориентированной инфраструктуры. Существенных объемов возврата инвестиций обычно не планируется.

Уровень 2 — повторяемый, ориентированный на Интранет (SOE repeatable projects — intra-focused). Требования данного уровня — наличие развитой СОА, в частности, использование одной или нескольких ESB. Должны быть специфицированы все задачи, которые могут быть решены средствами СОА. Определены все потенциальные пользователи КИС и зафиксированы их роли. Пристальное внимание уделяется вопросам качества, внедряются такие протоколы, как WS-Security, WS-ReliableMessaging. Основное внимание направлено на использование СОА-решений внутри организации.

ИТ-проекты направлены преимущественно на разработку и внедрение бизнес-процессов на базе созданных сервисов.

На этом этапе фактически начинается процесс возврата инвестиций. Основным источником является уменьшение стоимости внедрения отдельного бизнес-процесса за счет использования средств управления бизнес-процессами на основе BPEL и, в частности, за счет повторного использование бизнес-процессов.

Уровень 3 — базовый (SOE extended enterprise-focused). Основые задачи, решаемые на данном уровне, связаны с интеграцией ИТ-инфраструктуры организации в глобальное информационное пространство. Если на уровне 2 основным инструментом интеграции сервисов является оркестровка, то на уровне 3 — это хореография, т.е. объединение бизнес-процессов, протекающих в нескольких организациях. В качестве бизнес-целей выступает поиск бизнес-партнеров для организации совместного бизнеса. Реализуется мониторинг бизнес-процессов, протекающих в рамках нескольких организаций.

Основная тематика ИТ-проектов связана с организацией взаимодействия с партнерами по бизнесу. Возврат инвестиций достигается преимущественно за счет уменьшения стоимости поддержки партнерских связей.

Уровень 4 — базовый (SOE — solution focused). Основное внимание направлено на совершенствование сервисно-ориентированной инфраструктуры, которая обеспечивала бы эффективную реализацию как бизнес-процессов внутри организации, так и бизнес-процессов, в которых участвуют организации-партнеры.

Основные ИТ-проекты связаны с разработкой типовых горизонтальных и вертикальных решений, с помощью которых в короткие сроки при минимальных затратах можно реализовывать новые бизнес-процессы. Речь идет о широком внедрении «лучших практик», т.е. фреймворков.

Основным источником возврата инвестиций является снижение стоимости внедрения новых бизнес-процессов за счет использования типовых решений уровня фреймворков.

Уровень 5 — оптимизация производительности, гибкость, интеллектуализация (SOE – performance, agility, and intelligence focused). Ориентация идет на решение проблем, связанных с созданием гибких (agile) систем, способных эффективно адаптироваться к изменениям окружающей среды. Для этого активно используются механизмы работы с бизнес-правилами и политиками.

Внедряются механизмы автоматического обнаружения Web-сервисов, в частности, средствами семантического Web, построения бизнес-процессов в динамике.

Основные темы ИТ-проектов связаны с с внедрением интеллекуальных технологий, ориентированных на работу со знаниями, позволяющими эффективно адаптироваться к изменению требований, а основной источник возврата инвестиций — уменьшение суммарной стоимости владения системой в условиях быстро изменяющейся окружающей среды.

В заключение следует отметить, что рассмотренная SOE-модель зрелости во многом отличается от СОА-модели и ее можно рассматривать как расширение СОА.

Эти модели можно рассматривать как взаимодополняющие, при этом СОА модель — это видение системы с точки зрения интегратора, а SOE — с точки зрения менеджера [46].

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Перечислите основные типы взаимодействий в ИС.
2. Охарактеризуйте понятия синхронной и асинхронной связей.
3. Охарактеризуйте понятия сохранной и несохранной связей.
4. Перечислите и охарактеризуйте типовые подходы к интеграции приложений.
5. В каких случаях для интеграции приложений целесообразно использовать разделяемые базы данных?
6. В каких случаях для интеграции приложений целесообразно использовать удаленный вызов процедуры и методов?
7. В каких случаях для интеграции приложений целесообразно использовать механизмы, основанные на обмене сообщениями?
8. Для чего используется MPI?
9. Охарактеризуйте понятие системы, ориентированной на работу с сообщениями.
10. Что такое очереди сообщений?
11. Охарактеризуйте существующие модели обмена сообщениями точка-точка и публикация-подписка.
12. В чем суть интеграции приложений на уровне данных?
13. Что такое бизнес-функции и бизнес-объекты?
14. Что такое бизнес-процесс?
15. Что такое Web-сервисы?
16. Что такое BPEL?

17. Раскройте содержание понятий оркестровка и хореография Web-сервисов.
18. Каким образом в BPEL описываются асинхронное и синхронное взаимодействия?
19. Какова структура BPEL-документа?
20. Что такое партнерские связи?
21. Что такое бизнес-правила?
22. Охарактеризуйте системы управления бизнес-правилами.
23. Что такое портал и портлет?
24. Каково назначение порталов и какие выгоды можно получить от их использования?
25. Каким образом работает удаленный портлет?
26. Перечислите достоинства, недостатки и назовите области применения портлетов.
27. Охарактеризуйте общие принципы построения корпоративных сервисных шин.
28. В чем различие между A2A-интеграцией и B2B интеграцией?
29. Охарактеризуйте BizTalk Server.
30. Охарактеризуйте эталонную модель СОА, используемую в WebSphere.
31. Охарактеризуйте понятия уровнь зрелости сервисно-ориентированной архитектуры и сервисно-ориентированной организации.
32. Перечислите и охарактеризуйте основные уровни зрелости сервисно-ориентированной архитектуры.

Глава 7

АРХИТЕКТУРНЫЕ РЕШЕНИЯ РАЗРАБОТКИ ПРИЛОЖЕНИЙ

7.1. Подходы к архитектурным решениям корпоративных информационных систем

Любая КИС класса MRP/ERP рассчитана на множество реализаций и развитие, поэтому имеет огромное количество параметров настройки и строится как специализированный инструментарий для реализации проектных решений в области управления предприятием. Система всегда имеет свои развитые средства программирования и настройки. Проблема заключается в том, какие классы моделей (рис. 7.1) предприятия используются в качестве основы для настройки и какими средствами осуществляется такая настройка.

Настройку системы под конкретный проект внедрения желательно выполнять в рамках параметризации периода исполнения. Такие возможности позволяет обеспечить подход, основанный на динамическом описании модели предприятия и использовании ее в качестве основы для обеспечения нужных свойств конкретной реализации.

Настройка системы всегда опирается на модель предприятия. В настоящее время бизнес-модель предприятия часто строится в стандартах IDEF0, IDEF3, DFD и ER. К сожалению, не все про-

Рис. 7.1. Иерархия моделей в КИС

ектные решения по настройке системы могут быть адекватно описаны в рамках перечисленных моделей. Это ограничения моделей, а не инструментария. При создании корпоративных систем в качестве основы для описания проектных решений настройки предлагается использовать объектно-ориентированную модель предприятия. Основу такой модели составляет спецификация классов предметной области. Именно в понятиях этой модели требуется описать модель конкретного предприятия.

Такой подход базируется на идее классификации сущностей предметной области, которая выполняется в несколько этапов:

- выделение базовых классов предметной области, обладающих высокой степенью абстрагирования и стабильностью спецификации;
- разработка средств поддержки этих классов, как в СУБД, так и в приложении;
- разработка средств поддержки для динамического создания и спецификации подклассов в СУБД и в приложении;
- выделение производных классов предметной области применительно к каждому проекту внедрения индивидуально.

Данный подход предоставляет широкие возможности для построения схем классификации и взаимодействия с указанием заданных параметров классов, ориентированных на конкретное применение.

Разработка системы классов является предметом серьезной проектной работы на этапе разработки проекта внедрения и практически не требует поддержки и вмешательства группы программирования. Более того, работу по настройке может выполнять обученная группа внедрения от заказчика.

Для настройки многочисленных параметров последние сгруппированы по направлениям деятельности. Конфигурирование специализированных рабочих мест в соответствии с организационно-функциональной структурой, описанной в проекте внедрения, также осуществляется с использованием инструментальных средств настройки.

В качестве основных для моделирования в инструментальной подсистеме рассмотрим следующие абстрактные модели:

- статическая модель объектного представления предметной области;
- организационно-функциональная модель предприятия;
- модель деталей операций;
- модель функциональных семантических сетей;
- модель состояний;
- модель кооперации.

В настоящее время особый интерес представляют подходы к проектированию приложений на основе доменных архитектур. Работа с доменными архитектурами требует изменения в мышлении, по-

скольку при разработке новой системы программного обеспечения требуется классифицировать ее как систему одной или нескольких архитектурных областей [5]. Выбрав нужные классы архитектурных доменов можно использовать имеющиеся архитектурные решения для реализации конкретной информационной системы. Такой подход приводит к массовому повторному использованию архитектурных решений и дизайна, снижению риска неудач, поскольку эталонные модели проверены. Для описания предлагаемых решений будем использовать язык UML [1].

Каркасы проектирования. Каркас — это набор взаимодействующих классов, составляющих повторно используемый дизайн для конкретного класса программ [4]. Каркас диктует определенную архитектуру приложения. Он определяет общую структуру, ее разделение на классы и объекты, основные функции тех и других, методы взаимодействия объектов и классов и потоки управления. Эти параметры задаются каркасом, а прикладные проектировщики или разработчики могут сконцентрироваться на специфике приложения. В каркасе аккумулированы проектные решения, общие для данной предметной области. Акцент в каркасе делается на повторное использование дизайна, а не кода, хотя обычно он включает и конкретные подклассы, которые можно применять непосредственно.

Повторное использование на уровне каркасов меняет направление связей между приложением и программным обеспечением, лежащим в его основе, на противоположное. При использовании инструментальной библиотеки (или обычной библиотеки подпрограмм) пишут тело приложения и вызывают из него код, который планируют использовать повторно. При работе с каркасом, наоборот, повторно используют тело и пишут код, который оно вызывает. В результате приложение создается быстрее. Более того, все приложения имеют сходную структуру. Их проще сопровождать. С другой стороны, могут появиться ограничения на использовании тех или иных проектных решений.

Различия между паттернами и каркасами. Паттерны проектирования более абстрактны, чем каркасы [4]. В код могут быть включены целые каркасы, но только экземпляры паттернов. Каркасы можно писать на разных языках программирования и не только изучать, но и непосредственно исполнять и повторно использовать. В противоположность этому паттерны проектирования необходимо реализовывать всякий раз, когда в них возникает необходимость. Паттерны объясняют намерения проектировщика, компромиссы и последствия выбранного дизайна. Как архитектурные элементы паттерны проектирования мельче, чем каркасы.

Типичный каркас содержит несколько паттернов. Обратное утверждение неверно. Паттерны проектирования менее специализированы, чем каркасы. Каркас всегда создается для конкретной предметной области.

Значение каркасов возрастает. Именно с их помощью объектно-ориентированные системы можно использовать повторно в максимальной степени. Крупные объектно-ориентированные приложения составляются из слоев каркасов, взаимодействующих друг с другом. Дизайн и код в значительной мере определяются теми каркасами, которые применялись при его создании.

Согласно Робертсу (Roberts) и Джонсону (Jonson) каркасы воплощают теорию проблемных доменов (областей) и всегда являются результатом анализа этих доменов [4]. Каркас может быть адресован всей или только части архитектуры приложения.

Каркас определяется как множеством конкретных и абстрактных классов, так и определением способов их взаимоотношений. Конкретные классы обычно реализуют взаимные отношения. Абстрактные классы представляют собой точки расширения, в которых каркасы могут быть использованы и адаптированы. Точка расширения — это часть каркаса, для которого не приведена реализация.

Обычно каркас предоставляет только спецификацию интерфейса и точки расширения, а разработчик выбирает, как эти точки расширения должны быть адаптированы для реализации специфических изменчивых свойств конкретных приложений (линейки продуктов) [4].

Для подгонки архитектурных вариаций и обеспечения настраиваемости реализаций часто применяется параметризация. Могут также применяться и другие механизмы расширения: наследование классов и интерфейсов, переопределение методов и делегирование. Эти механизмы составляют точки изменчивости каркаса.

Рассмотрим архитектуру инструментальных средств, представленную в виде слоев каркасов, которые обеспечивают описание и реализацию проектных решений в терминах базовых метамоделей предметной области. Схема взаимосвязи моделей и их каркасов приведена на рис. 7.2.

Можно уменьшить степень сложности разработки, увеличить подвижность и жизнеспособность разрабатываемого продукта, повышая уровень абстракции для разработчиков. Одно из направлений — использование моделей для фиксации проектных решений в формах, легко поддающихся обработке. Этот подход называется разработкой, управляемой моделями MDD (Model-driven development) [4].

На рис. 7.2 представлена схема взаимосвязи моделей и их каркасов. Мета-метамодель позволяет описывать конкретные модели, используемые при проектировании информационных систем (ИС). Компонент Каркас описания метамоделей обеспечивает разработку и сопровождение базовых моделей, используемых при проектировании ИС. Следующий уровень моделей — это базовые Метамодели языков спецификаций. Каждая такая метамодель описывается средствами Каркаса описания метамоделей.

Проектные решения описываются в соответствии с правилами выбранных Метамоделей языков спецификаций. Компоненты Каркасы

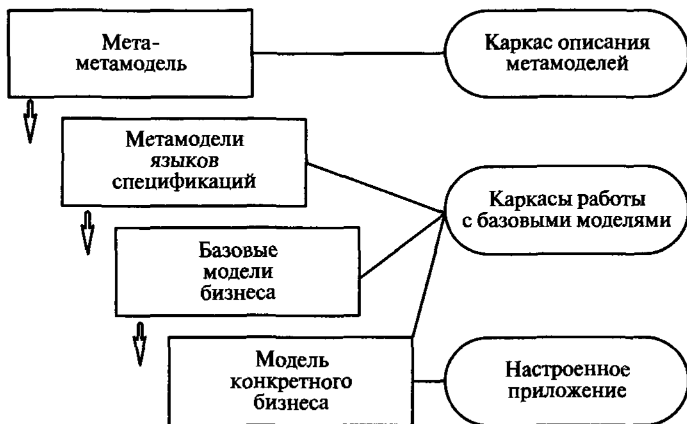


Рис. 7.2. Схема взаимосвязи моделей и их каркасов

работы с базовыми моделями предоставляют необходимые сервисы для фиксации и исполнения проектных решений. Каждый такой каркас должен иметь необходимый набор методов, обеспечивающих качественное решение прикладных задач в соответствующем домене предметной области. Описание в терминах конкретной модели обеспечивает необходимый уровень расширения при реализации конкретного приложения. Следующий уровень это Базовые модели бизнеса, описанные в терминах выбранных языков спецификаций. Каждая такая модель должна иметь необходимый набор точек расширения для конкретизации модели конкретного бизнеса.

Последний уровень — Модель конкретного бизнеса, которая отражает результаты конкретного проекта внедрения приложения.

Выбор базовых моделей для построеиия КИС. Вариант технологии разработки приложения на основе базовой типовой модели предметной области приведен на рис. 7.3.

На основе использования Абстрактных моделей разрабатывается проект ИС, основу которой составляет Базовая модель ПО. Используя Внешние инструментальные средства осуществляется реализация проекта. Результатом реализации являются Метаданные базовой модели и Код ядра приложения. Базовая модель может быть построена в виде каркасов для конкретных доменов предметной области.

Конкретное приложение разрабатывается также с использованием Абстрактных моделей и Базовой модели ПО. Результатом проектирования является Проект внедрения, который опирается на Базовую модель ПО и Метаданные базовой модели. Реализация Проекта внедрения выполняется путем расширения Метаданных базовой модели и Кода ядра приложения. В результате генерируется Код проекта внедрения. На их основе генерируется уникальный для проекта внедрения Исполнитель приложения.

Разработка проекта КИС

Проект базовой модели ПО

Внешние инструментальные средства

Реализация КИС

Абстрактные модели

Метаданные базовой модели ПО

Код ядра приложения

Разработка проекта внедрения

Реализация проекта внедрения

Встроенные инструменты. Средства периода компиляции

Проект внедрения

Метаданные проекта внедрения

Код проекта внедрения

CASE-средства проекти-рования

рования

Исполнитель приложения

Рис. 7.3. Схема реализации проектных решений на основе базовой модели ПО

Вариант технологии разработки приложения на основе моделирования абстрактных моделей показан на рис. 7.4. В дополнение к построению базовой модели предметной области данный вариант предполагает строить архитектуру системы на основе моделей более высокого уровня абстрагирования, а именно на основе поддержки моделей, средствами которых могут быть описаны модели конкретных проблемных доменов. Результатом такого проекта является Проект моделирования АМ, в котором средствами АМ частично конкретизированы решения для базовой модели ПО. В результате реализации такого проекта будут получены Метаданные для моделирования АМ и Код ядра для работы с АМ, который обеспечивает необходимый набор сервисов для работы с объектами базовой модели ПО и Инструмент настройки, который обеспечивает формирование Модели проекта в терминах АМ путем расширения базовой модели ПО.

Проект внедрения разрабатывается так же, как и в предыдущем варианте, однако реализация проекта существенно отличается. Реализация проекта внедрения осуществляется с использованием Инструментов настройки путем конкретизации Метаданных базовой

Рис. 7.4. Схема реализации проектных решений на основе моделирования абстрактных моделей

модели средствами формирования Модели проекта в терминах АМ. В результате формируется Модель проекта в терминах АМ. При этом все проектные решения проекта внедрения, трансформирующиеся в Модель проекта в терминах АМ, являются данными Метаданных для моделирования АМ и интерпретируются общим для всех проектов Исполнителем приложения.

Если сервисов, поддерживающих АМ и базовую модель ПО, недостаточно для реализации конкретного приложения, то придется расширять проект базовой модели уже средствами периода компиляции.

Язык моделирования, специфичный для домена, DSL. Модель становится видом исходного кода, позволяя разработчику манипулировать проектными решениями в графическом виде или с использованием специального пользовательского интерфейса, управляемого синтаксисом языка моделирования. Ценность инструмента увеличивается благодаря раскрытию отношений и зависимостей, которые трудно увидеть в исходном коде, а также подавлению несущественных деталей.

Удержание модели на высоком уровне абстракции можно обеспечить путем приближения платформы к модели. Этот подход может быть реализован с использованием каркасов или трансформаций. Можно использовать каркас для реализации абстракций, появляющихся в моделях, и применять модели для дополнения и завершения точек расширения в каркасах. Проектные решения, выраженные средствами моделей, помогают дополнить каркас.

В качестве представительного примера для изучения архитектур ИС будем рассматривать проблемный домен информационной поддержки жизненного цикла изделий (ЖЦИ).

Для построения, внедрения и сопровождения линейки продуктов для решения задач управления ЖЦИ рассмотрим следующий путь:

1) разработка высокоуровневых каркасов, поддерживающих метамодели проектирования ИС управления жизненным циклом изделий (ЖЦИ);

2) выделение проблемных доменов, покрывающих область ЖЦИ;

3) разработка системы взаимосвязанных каркасов, реализующих выделенные проблемные домены;

4) разработка инструментов работы с точками расширения каркасов.

Взаимное соответствие моделей спецификации. Рассмотрим следующие варианты моделей спецификации:

- текстовое представление на специализированном языке;
- графическое представление;
- XML-представление;
- реляционное представление.

Текстовое представление на специализированном языке. Разработка специализированного языка для описания проектных решений для проблемного домена является сложным и дорогостоящим решением. Однако наличие соответствующих каркасов позволит не отвлекаться в спецификации на ключевые решения, принятые при реализации каркаса, а сосредоточиться на проблемах использования сервисов каркаса и конкретизации его точек расширения средствами специализированного языка. При этом компоненты высокоуровневой модели каркасов должны быть явно вплетены в синтаксис и семантику соответствующего языка. Вопрос о внутреннем хранении спецификации требуется решать отдельно.

Графическое представление. Графическое представление высокоуровневых спецификаций (например, на языке UML [5, 61]) обладает рядом преимуществ по сравнению со специализированными доменными языками. Прежде всего спецификация используемых каркасов представляется в наглядной и понятной форме. Большинство точек расширения связано с конкретизацией абстрактных классов каркаса. Поэтому конкретизация в большинстве случаев также может быть выражена графическими средствами

в контексте абстрактной спецификации каркаса. Вопрос о внутреннем хранении спецификаций может быть решен с использованием XML-форматов. UML-спецификация может быть трансформирована в XML-спецификацию.

XML-представление. Язык спецификации структур XML является универсальным средством для описания любых структурных моделей. Инструменты, поддерживающие работу с XML-спецификациями, достаточно развиты и предоставляют все необходимые возможности манипулирования и исполнения высокоуровневых спецификаций. Для эффективного использования спецификации каркасов должны быть описаны средствами XML, что обеспечит их правильное использование при конкретизации проектных решений. XML-представление можно трансформировать в ER-представление, и наоборот.

Реляционное представление. Реляционная модель является универсальной как для спецификации разнообразных информационных структур (метаданных), так и для ведения конкретных данных в соответствии с этими метаданными. Реляционная модель поддержана развитыми средствами спецификации и доступа к данным. Особо следует отметить высокое качество поддержки целостности данных, обеспечиваемое всеми производителями реляционных СУБД. Однако вопрос о представлении данных на пользовательском уровне требует разработки специальных приложений. Проблемы хранения данных и доступа к данным, эффективно решаемые в среде реляционного представления, делают это представление наиболее предпочтительным для целей внутреннего хранения спецификаций каркасов и их расширений.

Открытый набор каркасов может быть реализован в среде СУБД и инструментальными средствами общего назначения (в частности, в среде DELPHI 7). Высокоуровневые спецификации каркасов для выделенных доменов поддерживаются с использованием базового каркаса ИС.

Управление схемами классификации информационных объектов. Реализация каждого каркаса выполняется как в среде СУБД, так и в среде пользовательского приложения.

В среде СУБД для каркаса разрабатывается модель хранения данных, процедуры поддержки целостности данных (Model), высокоуровневые процедуры манипулирования данными и специализированные процедуры обработки данных.

В среде инструментальных средств общего назначения (например, DELPHI 7) разрабатывается пользовательский интерфейс для доступа к сервисам каркаса (модули View и Controller).

Высокоуровневые спецификации могут разрабатываться в различных инструментальных средах. Конструктор моделей позволяет трансформировать спецификации, представленные в виде XML-спецификаций, во внутренний ER-формат в контексте спецификаций

имеющихся каркасов. Инструментальный модуль ИС, обычно входящий в состав системы управления ИТ-проектами ИС, предоставляет необходимые сервисы для работы с высокоуровневыми спецификациями каркасов (визуальное представление, конкретизация в соответствующих точках расширения, документирование, проверка корректности, полноты, непротиворечивости).

Для построения линейки продуктов необходимо выполнить следующие работы:

1) провести анализ процессов управления ЖЦИ для выявления требований к сервисам информационной поддержки;

2) выделить те требования, которые в настоящее время не поддержаны ИС;

3) разработать проект расширения функционала ИС и способы его реализации.

Пример иерархической структуры каркасов для решения задач управления ЖЦИ приведен на рис. 7.5. Основным является Каркас ООМ, обеспечивающий моделирование объектных представлений в среде реляционных СУБД и решение задач построения и представления данных объектной модели бизнеса. На следующем уровне располагаются Каркасы общего назначения, которые обеспечивают

Рис. 7.5. Иерархия каркасов, подразделяющих ЖЦИ

работу с базовыми моделями, используемыми при описании различных аспектов управления деятельностью. К ним относятся каркасы: Бизнес-логика, Сети вычислений, Отчеты, Интеграция, Пользовательский интерфейс. Каждый из этих каркасов позволяет решать свои задачи в рамках соответствующих базовых моделей.

Следующий уровень — каркасы базовых объектов — позволяет моделировать свойства и методы базовых объектов области управления ЖЦИ. К ним относятся каркасы: Изделие, Производственные мощности, Трудовые ресурсы, Документ, Хозяйственные операции. Практически любые процессы ЖЦИ могут быть эффективно смоделированы на основе использования перечисленных выше каркасов. При этом каркас ООМ обеспечивает возможности конкретизации объектов для любого из каркасов, расположенных ниже.

На последнем уровне располагаются каркасы управления ресурсами, которые позволяют моделировать процессы ЖЦИ. К ним относятся каркасы: Продажи, Выпуск, Остатки, Изготовление, Закупки, Сопровождение, Бюджетирование, Другие процессы.

7.2. Моделирование структуры классов и их свойств

Классификация объектов предметной области не однозначна и определяется точкой зрения заинтересованных сторон. Особенно это проявляется при выборе проектных решений для информационной поддержки жизненного цикла изделий (ЖЦИ). Поэтому каркас для спецификации объектов предметной области должен обеспечить функциональные возможности моделирования разнообразных схем классификации одних и тех же объектов в соответствии с разными точками зрения. В качестве заинтересованных сторон могут выступать организации, контролирующие соблюдение ГОСТов [42], различные группы пользователей, формулирующие свои требования в соответствии с особенностями доменов проблем, в которых они работают. Наиболее важным и наиболее сложным для моделирования информационным объектом систем управления ЖЦИ является информационный объект Изделие. Поэтому некоторые проблемы построения архитектуры приложения будем рассматривать на примере структуры каркасов для работы с данными об изделиях.

Каркасы моделирования метаданных. Введем несколько определений.

Объект (от лат. *objectum* — предмет) — то, что существует вне нас и не зависимо от нашего сознания (внешний мир, действительность) и является предметом познания, практического воздействия.

Информационный объект (*ИО*) — совокупность данных и программного кода, обладающая свойствами (атрибутами) и методами, позволяющими определенным образом обрабатывать данные. Са-

мостоятельная единица применения и хранения в интегрированной информационной среде (ИИС) [2].

Сущность — совокупность всех необходимых сторон и связей (законов), свойственных вещи и взятых в их естественной взаимозависимости. Сущность отличается от явления, которое есть обнаружение сущности через свойства и отношения, доступные чувствам.

Сущность ERD — стереотип модели сущность-связь (ER), предназначенный для представления объектов реального мира в концептуальной ER-модели.

Класс-сущность — стереотип модели UML, является потомком метаобъекта.

Класс — предназначен для представления объектов реального мира в концептуальной UML-модели.

Элемент — экземпляр класса (сущности).

Статическая модель объектного представления предметной области является базовой для поддержки других моделей.

Для многих приложений ИС объект Изделие является главным и определяющим. Естественно, архитектура ИС таких приложений во многом определяется информационной моделью такого объекта.

Проблемы классификации объекта Изделие. Существует множество признаков, которые могут быть использованы в качестве оснований для классификации объекта Изделие. Классификация представ-

Рис. 7.6. Фрагмент схемы классификации объекта Изделие

ляется в виде схемы классификации, фиксирующей конкретизацию (обобщение) ИО Изделие. Шаг конкретизации предполагает выбор соответствующих оснований классификации, каждое из которых задает альтернативные конкретизации. Полная схема классификации может быть разложена на отдельные подсхемы — деревья. В подсхеме подкласс конкретизируется только по одному основанию классификации. Назовем каждую такую схему альтернативной схемой классификации. Фрагмент многоаспектной схемы классификации ИО Изделие показан на рис. 7.6. Как видно из рис. 7.6, уровень абстрагирования при построении модели изделия достаточно высок.

Конкретизация ИО Изделие проведена по двум основаниям Физическая природа и Уровень стандартизации. По первому основанию выделены следующие подклассы: Вещество, Материальный предмет, Услуга, Программная система.

По второму основанию выделены следующие подклассы: Стандартное изделие, Нормализованное изделие, Оригинальное изделие.

В свою очередь, подкласс Материальный предмет конкретизируется по крайней мере по двум основаниям: По структуре, По функциональному назначению. Каждая ветвь конкретизации отражает точки зрения соответствующих заинтересованных сторон. Выделение класса влечет за собой разработку атрибутивных моделей и набора специфических методов.

Введем ИО Точка зрения. Будем считать, что каждой Точке зрения соответствует своя безальтернативная схема классификации. Каждая точка зрения поддерживается одной или несколькими Заинтересованными сторонами.

Выделяя некоторый класс, мы предполагаем следующее:

- наследование свойств и методов родительского класса;
- наличие собственных дополнительных свойств и методов класса-потомка.

Каждая Точка зрения может предполагать следующее:

- выбор собственного основания для конкретизации некоторого класса со своими собственными дополнительными свойствами и методами новых введенных подклассов;
- добавление свойств и методов классов в уже имеющуюся схему конкретизации.

Имеющееся разнообразие возможных схем классификации изделий не дает возможности выбрать какую-либо одну из них для формирования каркаса. Поэтому для построения каркаса Изделия объектом моделирования сделаем собственно схему классификации. Фрагмент модели для решения задач описания многоаспектной классификации приведен на рис. 7.7.

Сущность Класс изделия позволяет создавать и специфицировать наборы классов изделий. Для каждого абстрактного класса в соответствии с точками зрения вводятся различные основания классифи-

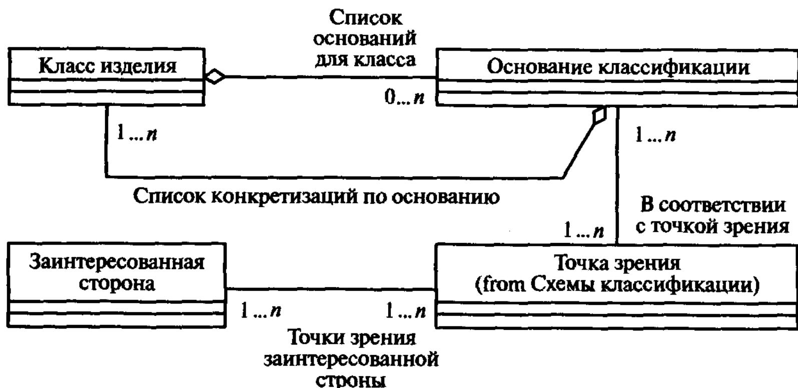


Рис. 7.7. Концептуальная схема для моделирования вариантов классификации изделий

Метакласс
(from Метаклассы КИС)
Базовый класс
(from Метаклассы КИС)
1
Родитель
1
Потомок
Конкретизация
класса
0...n
0...n
Схема
классификации
Основание
классификации
0...n
Категория
(from Метаклассы КИС)
1
0...n
Виртуальный класс
(from Метаклассы КИС)
Для точек
зрения
1...n
Точка зрения

Рис. 7.8. Расширение каркаса для моделирования альтернативных схем классификации

кации (сущность Основание классификации и ассоциации Список оснований для класса, В соответствии с ТЗ). Для каждого основания классификации соответствующего абстрактного класса требуется задать список классов конкретизации (ассоциация Список конкретизации по основанию). Для отражения источников точек зрения введена сущность Заинтересованная сторона и ассоциация Точки зрения заинтересованной стороны.

Расширение представленного решения на любые информационные объекты приведено на рис. 7.8.

Сущность Метакласс позволяет ввести реестр всех стереотипов и схем их конкретизации. Сущность Схема классификации и ассоциация Конкретизация ИО позволяют описывать разнообразные варианты конкретизации ИО предметной области. Ассоциации Потомок и Основание классификации позволяют указать альтернативные элементы схемы классификации. Ассоциация Для точек зрения позволяет указать точки зрения, которые отражает вариант классификации.

Проблема моделирования свойств ИО. Информационная поддержка функций работы со справочником изделий не ограничивается проблемами классификации. Выделение класса сопровождается выделением дополнительных атрибутов, присущих данному подклассу изделий. По этой причине архитектурное решение для работы со справочником изделий должно обеспечить удобное моделирование разнообразных свойств различных подклассов объекта Изделие.

Концептуальная схема для моделирования свойств информационных объектов показана на рис. 7.9.

Сущность Метакласс позволяет вести каталог классов предметной области. Сущность Атрибут класса позволяет вести каталог атрибутов для спецификации свойств объектов. Ассоциативная сущность Атрибут в составе ассоциации Состав атрибутов, Исполнительный атрибут позволяют моделировать спецификации свойств вновь вводимых классов. Для задания свойств сущности Атрибут класса вводятся следующие ассоциации:

- Тип атрибута задает тип атрибута;
- Вариант множественности задает вариант множественности атрибута;
- Вариант агрегации задает вариант агрегации атрибута;
- Инверсный атрибут указывает на атрибут, являющийся инверсным для заданного атрибута.

Так как сущность Атрибут класса является подклассом сущности Метакласс, то для нее могут быть добавлены специальные дополнительные характеристики и схемы классификации.

В результате настройки диаграммы классов на уровне исполнения появляется возможность работать с каталогами элементов любого из классов.

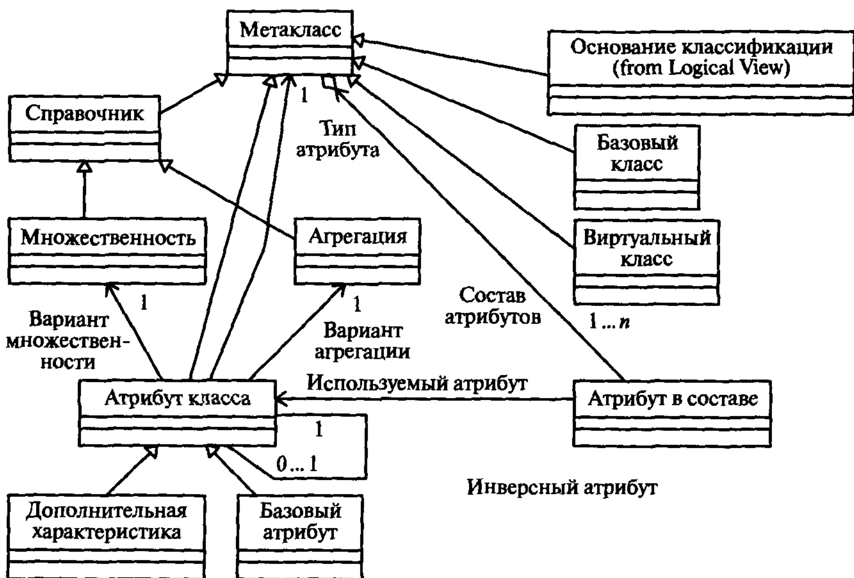


Рис. 7.9. Концептуальная схема для моделирования свойств объектов

Модель кооперации для спецификации бизнес-логики. Бизнес-логика является одной из быстроменяющихся частей приложения. Традиционные инструменты программирования позволяют решить любые задачи ее реализации. Однако остается разрыв между спецификацией логики в проекте и ее программной реализацией. Этот разрыв заполняется кропотливой совместной работой специалиста предметной области, проектировщика и прикладного программиста. Одна из проблем спецификации логики — декларативная часть описания данных, на которых определены соответствующие логические операции. Модели спецификации потока управления не могут быть исполнены без спецификации соответствующих данных. Рассмотрим возможности, которые предоставляет модель кооперации UML. Основу кооперации составляет ролевая структура, на роли в которой должны быть назначены экземпляры соответствующих классов. Активизация экземпляра кооперации требует заполнить ролевую структуру соответствующими экземплярами элементов, которые будут двигаться по своей траектории и взаимодействовать между собой, выполняя тем самым предусмотренную спецификацией логику кооперации. Для спецификации логики бизнес-процессов на пользовательском уровне за основу возьмем описанную выше модель с некоторыми ограничениями:

- траекторию движения в соответствии с ролью будем описывать в терминах автоматной модели как последовательность поименованных состояний;

- учитывая, что в большинстве случаев переход в новое состояние определяется решением пользователя, свяжем с каждым состоянием множество допустимых поименованных решений;
- основу ролевой структуры будут составлять атрибуты, соответствующие ассоциациям между классами предметной области;
- с каждым состоянием свяжем подмножество доступных бизнес-функций, подмножество пользователей, которым разрешен доступ к функциям и решениям;
- с каждым состоянием свяжем предикативную функцию активизации, определенную на множестве состояний и допустимых решений элемента текущего класса, множестве состояний и допустимых решений элементов других классов, связанных ролевой структурой, множестве доступных функций, множестве значений доступных функций.

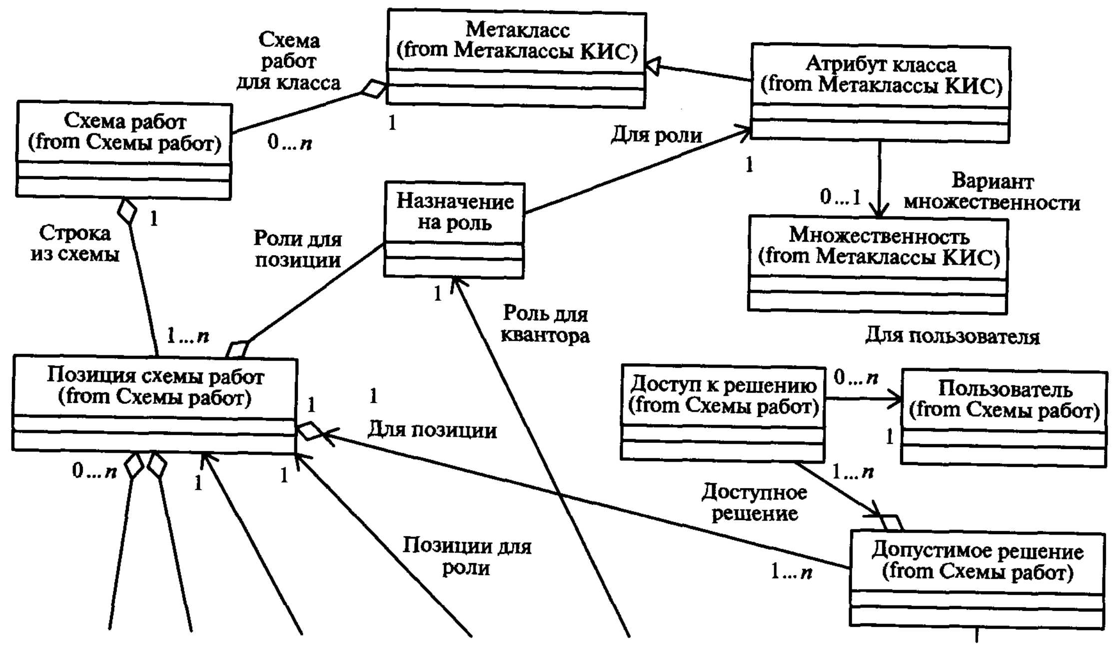
Возможность расширения схем классификации путем введения новых подклассов и ассоциаций совместно с описанной моделью кооперации позволяет моделировать сложную бизнес-логику непосредственно на пользовательском уровне.

Рассмотрим один из вариантов более подробной спецификации данной модели. На рис. 7.10 представлена диаграмма классов, отражающая проектные решения логического уровня для решения задач спецификации бизнес-логики с использованием ограниченной модели кооперации. Как видно из рис. 7.10, с каждым классом предметной области можно связать несколько элементов сущности Схема работ (ассоциация Схема работ для класса). Каждая схема описывает конкретный вариант поведения соответствующих элементов. Для описания состава состояний автоматной модели и поведения элементов класса вводится сущность Позиция схемы работ и ассоциация Строка из схемы. Спецификация позиции схемы включает следующие компоненты:

- множество допустимых решений (сущности Решение, Допустимое решение и ассоциации Разрешенное решение и Для позиции);
- функцию активизации (сущность Конъюнкция и ассоциация Входное условие);
- список точек, в которых должны быть назначены элементы на роли (сущность Назначение на роль и ассоциации Роль для позиции и Для роли);
- список доступных бизнес-функций (сущности Бизнес-функция, Доступная функция и ассоциации Работа для состояния, Функция в список).

7.3. Поддержки функций приложения

Полномасштабное проектирование — это реализация согласованной детальной программы проектирования, включающей создание



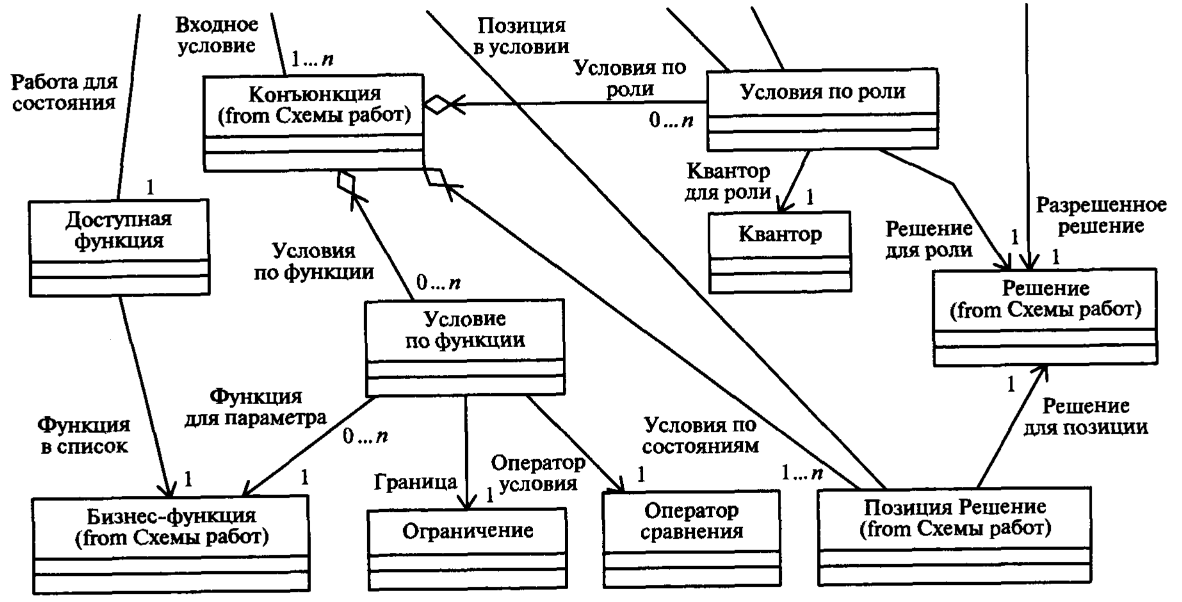


Рис. 7.10. Диаграмма классов для спецификации бизнес-логики

моделей или опытных образцов и проведение необходимых испытаний, целью которых является постановка изделия на производство и ввод его в эксплуатацию с разработкой всей необходимой документации и обеспечением информационной поддержки жизненного цикла изделия. Примерный состав каркасов, поддерживающих решение задач управления жизненным циклом изделий (ЖЦИ), был представлен ранее (см. рис. 7.5).

В качестве примера рассмотрим каркасы для работы с данными об изделии и каркас для организационно-функциональной структуры предприятия.

Каркасы для работы с данными об изделии. Конструкторские данные. Базовое представление об изделии (в соответствии 3.2.26 ГОСТ Р ИСО 10303-1) можно описать в виде диаграммы классов, в которой выделены основные информационные объекты и их наиболее важные взаимосвязи (рис. 7.11).

Основной сущностью является Изделие. В качестве основы для классификации изделий могут быть приняты различные основания (аспекты рассмотрения) [4]. На диаграмме для классификации принят аспект, характеризующий природу изделия. Видно, что стандарт определяет изделие достаточно широко. Соответственно к каждому из подклассов изделий применим подход моделирования, основанный на понятии жизненного цикла. Таким образом, высокий уровень абстрагирования понятия Изделие позволяет применять данный подход к широкому спектру объектов в различных областях деятельности. Основой эффективного управления ЖЦИ является применение информационных технологий поддержки. В свою очередь, применение информационных технологий поддержки ЖЦИ приводит к необходимости создания интегрированной информационной среды (ИИС).

Для описания конструкторской спецификации (КС) изделия введены сущность Строка КТС и ассоциации Строка для изделия и Входящее изделие. Для решения задач управления изменениями КС введена ассоциация База для изменения. Для учета модификаций изделий введена ассоциация База для модификации. Для учета допустимых замен в составе изделия введена ассоциация Замена для строки. Описанные компоненты модели позволяют моделировать только наиболее простые варианты использования приложения для работы с данными об изделии. Одним из наиболее трудных для моделирования является процесс управления конфигурацией изделия.

Управление конфигурацией — это процесс, включающий в себя: идентификацию, проверку, изменение конфигурации, подготовку отчетности об этих действиях. Для решения задач управления конфигурацией модель должна обеспечить представление большого количества допустимых вариантов исполнения для конкретного изделия.

Часть свойств изделия определяется как Параметр конфигурации. Каждый набор допустимых значений параметров конфигурации задает условия для формирования структуры изделия, соответствующей

заданному варианту исполнения. Для этой цели введена сущность Строка параметров настройки и ассоциации Список параметров настройки, По параметру. По мере необходимости создаются элементы сущности Вариант конфигурации. Связь с элементами Продукция обеспечивается ассоциацией Для продукции. Для каждого элемента Вариант конфигурации должны быть заданы допустимые значения соответствующих параметров конфигурации. С этой целью в модель введены сущность Значение параметра и ассоциации Для варианта, Значение, По параметру. Область значений каждого параметра задается с использованием сущности Допустимое значение параметра и ассоциации Ограничение. Для того, чтобы собирать спецификацию нужного варианта Изделия, необходимо с нужными элементами сущности Строка КТС связать логическую функцию, которая обеспечит управление вхождением строки в спецификацию конкретного варианта конфигурации, заданного значением соответствующих параметров конфигурации. С этой целью в модель введена сущность Функция конфигурации и ассоциация Функция для строки. Функция конфигурации определена на значениях Параметров конфигурации. Каждый Элемент функции задан как ограничение на значение соответствующего элемента сущности Параметр конфигурации (ассоциации Параметр для функции, Ограничение для функции). Таким образом, активность каждой строки спецификации на всю глубину детализации определяется соответствующими функциями конфигурации, заданными на множестве значений параметров конфигурации.

Представленное проектное решение является основой для реализации модуля ведения конструкторских данных об изделии.

Технологические данные. Для работы с технологическими данными об изделии информационная система должна обеспечить выполнение следующих требований:

• разработку технологических процессов, поддержку разных вариантов организации технологической подготовки, а именно с расцеховкой, без предварительной расцеховки, разработку сквозных техпроцессов, коллективную разработку техпроцессов несколькими технологами;

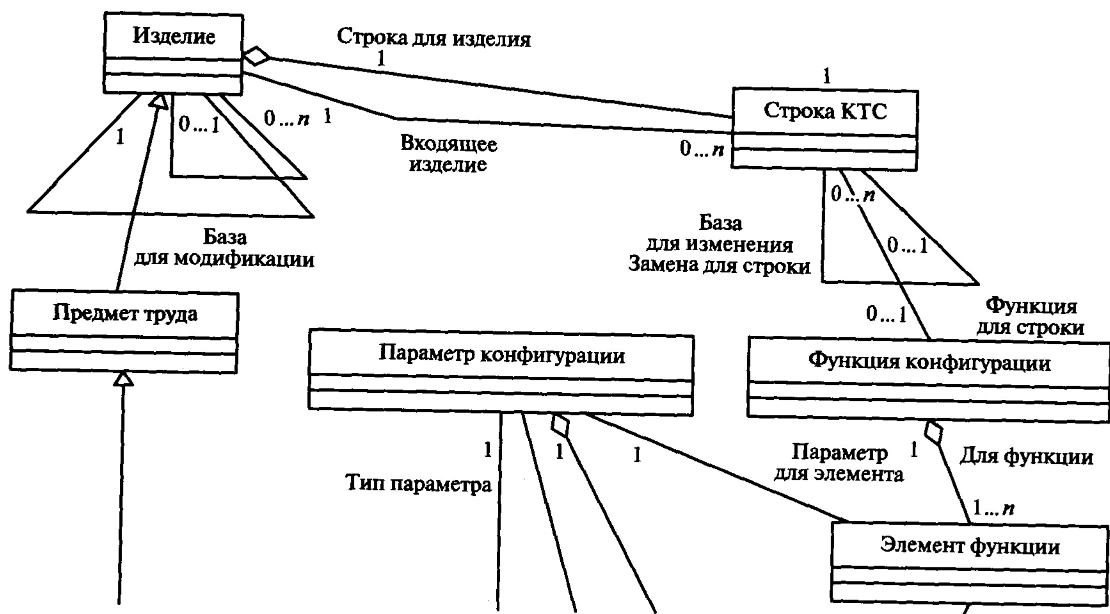
• разработку техпроцессов с разной степенью детализации (маршрутную, маршрутно-операционную, операционную технологии);

• разработку нескольких альтернативных техпроцессов изготовления для одной детали;

• настройку на разные режимы разработки техпроцессов (диалоговый, с использованием «мастеров» проектирования, по аналогу, из фрагментов ТП других деталей, на основании ТП комплексной детали);

• настройку системы для автоматизации проектирования ТП (создание и использование пользовательских функций для подбора оборудования и инструмента, генерации текстов переходов и другое);

• автоматизированное формирование комплектов технологической документации разных назначений и степени сложности;



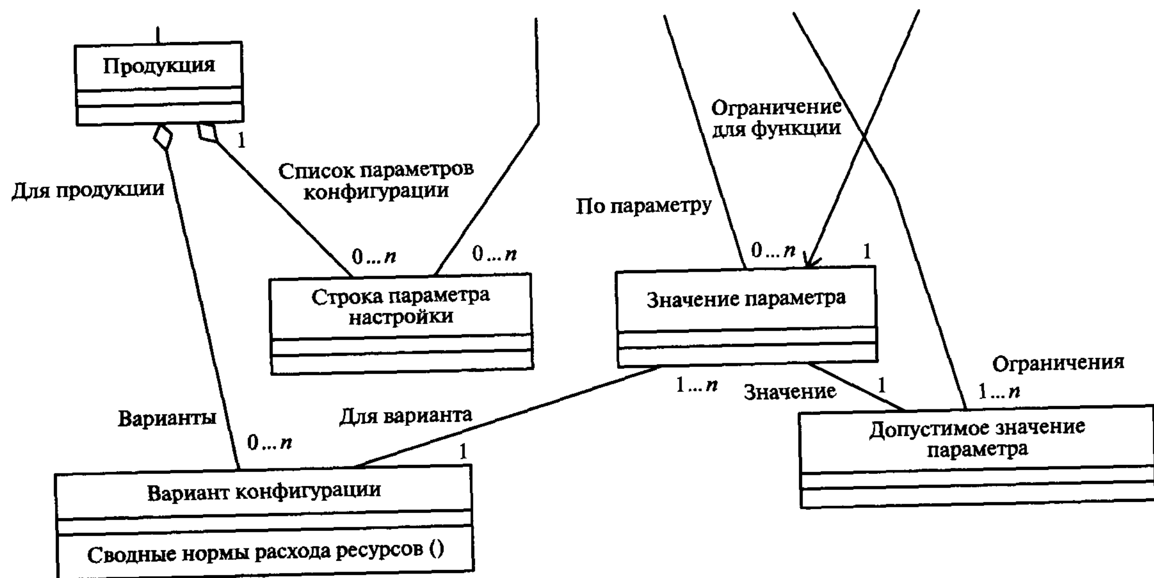
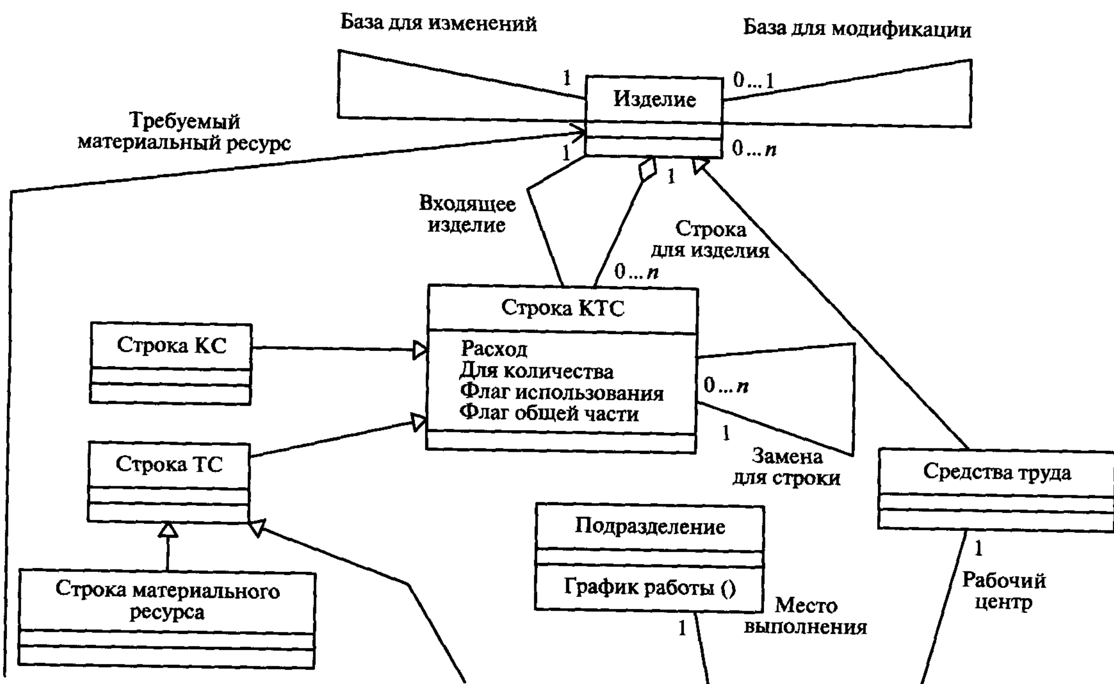


Рис. 7.11. Проектное решение задачи Управление конфигурацией



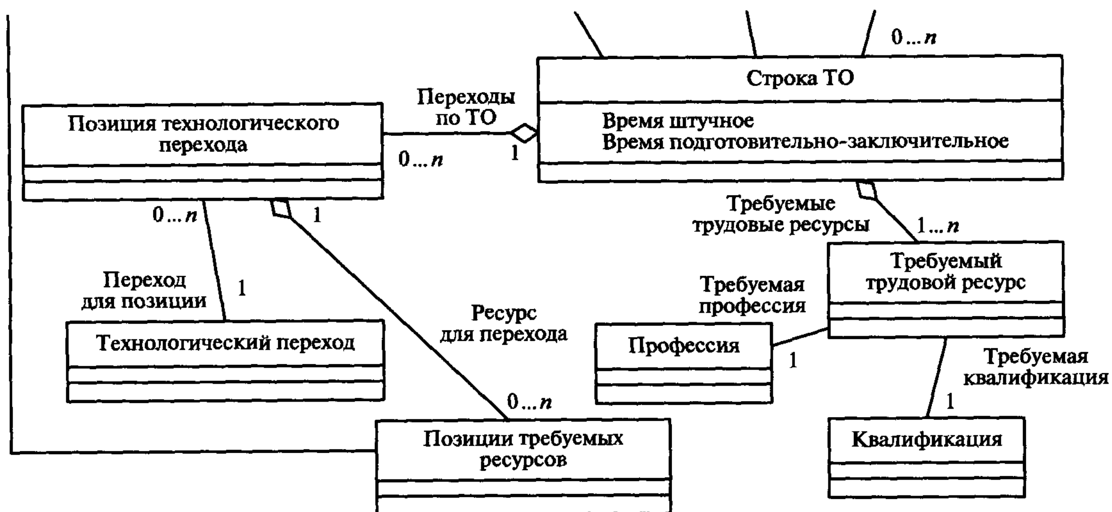


Рис. 7.12. Диаграмма классов проектного решения моделирования технологических спецификаций

- интеграцию с САМ-системами;
- создание единой БД технологических процессов и решений, справочника типовых элементов технологических процессов для быстрого проектирования ТП на новые детали;
- ведение электронной картотеки средств оснащения (инструмент, приспособления, оснастка с указанием характеристик, связи с чертежами, отслеживание применяемости);
- автоматизированный расчет массы заготовки и норм расхода, возможность подключения собственных алгоритмов для расчета норм расхода;
- ведение норм расхода вспомогательных материалов, норм расхода по типовым и групповым техпроцессам (нанесение покрытий, сварка т.д.);
- автоматическое формирование различных сводных материальных ведомостей (специфицированных, подетальных, по цехам, по изделиям);
- ведение БД материальных нормативов в соответствии с применяемыми технологическими процессами изготовления.

Особенностью диаграммы классов проектного решения, представленной на рис. 7.12, является введение сущности Строка КТС, позволяющая единообразно описать как конструкторскую, так и технологическую спецификацию Изделия (конструкторско-технологическая спецификация — КТС) [69].

Атрибуты Расход, Для количества, Флаг использования, Флаг общей части позволяют задать нормы расхода компонентов для соответствующего изделия.

Классификация строк КТС обеспечивает их разделение по назначению. Выделены следующие подклассы: Строка КС (строка конструкторской спецификации) и Строка ТС (строка технологической спецификации).

В свою очередь, сущность Строка ТС конкретизирована следующим образом: Строка материального ресурса (строка для описания норм расхода сырья, материалов и покупных комплектующих); Строка ТО (строка для описания технологической операции).

Сущность Строка материального ресурса позволяет ввести данные о нормах расхода основных и вспомогательных материалов. Строки технологических операций (сущность Строка ТО) позволяют описывать последовательность обработки изделия (маршрутная технология) и ввести нормы времени (атрибуты Время штучное, Время подготовительно-заключительное). Детализация каждой технологической операции (операционная технология), при необходимости, может быть описана с использованием сущности Позиция технологического перехода и ассоциации Переходы по ТО. Введение реестра параметризованных технологических переходов поддерживает сущность Технологический переход. Ассоциация Переход для позиции позволяет конкретизировать переходы при

описании Строки ТО. Требуемые материальные ресурсы для перехода позволят описать сущность Позиция требуемых ресурсов и ассоциации Ресурс для перехода и Требуемый материальный ресурс. Трудовые ресурсы для выполнения технологической операции позволяют описать сущности Требуемый трудовой ресурс, Профессия, Квалификация и ассоциации Требуемые трудовые ресуры, Требуемая профессия и Требуемая квалификация. Требуемый ресурс оборудования позволяет описать ассоциацию Место выполнения, указывающую на Подразделение, и ассоциации Рабочий центр, указывающую на требуемое Средство труда для выполнения технологической операции.

Каркас для организационно-функциональной структуры предприятия. Организационно-функциональная структура предприятия описывает структуру организационных звеньев предприятия, структуру бизнес-процессов предприятия и структуру исполнения, отражающую взаимосвязи организационных и функциональных элементов между собой [4]. На рис. 7.13 приведен пример классификации такой сущности как Оргзвено. Ассоциация Входит в состав позволяет описать иерархическую структуру подчиненности организационных звеньев. Конкретизация сущности Оргзвено выполнена в соответствии со степенью детализации. На первом уровне конкретизации выделено два подкласса: Контрагент, Подразделение.

Сущность Подразделение в соответствии с формой организации конкретизируется следующим образом: Отдел, Цех, Склад, Лаборатория, Участок, Служба.

Несомненно, приведенная конкретизация не является исчерпывающей. В каждом конкретном применении подобная схема должна настраиваться индивидуально.

Рис. 7.13. Пример классификации организационного звена

Структура бизнес-процессов предприятия связывает бизнес-процессы между собой, оргзвеньями и документами. Модель бизнес-процессов предприятия является важной частью информационной поддержки деятельности [4]. На рис. 7.14 представлен пример концептуальной модели, поясняющий взаимосвязи бизнес-процессов, подразделений и поддерживающих их документов. Основной сущностью данной схемы является Бизнес-процесс. Структура бизнес-процессов моделируется путем введения сущности Компонент бизнес-процесса и ассоциации Состав бизнес-процесса. Ассоциация Используемый бизнес-процесс позволяет указать входящий бизнес-процесс, соответствующий компоненту. Сущность Роль оргзвена в бизнес-процессе и ассоциация Участники бизнес-процесса позволяют задать роли каждого подразделения в каждом бизнес-процессе. Ассоциация Исполнитель бизнес-процесса указывает на роль конкретного оргзвена в бизнес-процессе. Сущность Роль оргзвена и ассоциация По роли позволяют задать конкретные роли для каждого бизнес-процесса. Для описания связей документов, бизнес-процессов и оргзвеньев в аспекте моделирования организационно-функциональной структуры введены следующие элементы. Сущность Роль документа в бизнес-процессе и ассоциация Используемые документы позволяют задать роли документов в каждом бизнес-процессе. Ассоциация Для документа позволяет указать конкретный документ на роль. Ассоциация Исполнитель по роли позволяет задать конкретное оргзвено по операции с документом. Сущность Операция описывает набор типовых операций с документами. Ассоциация По операции позволяет указать

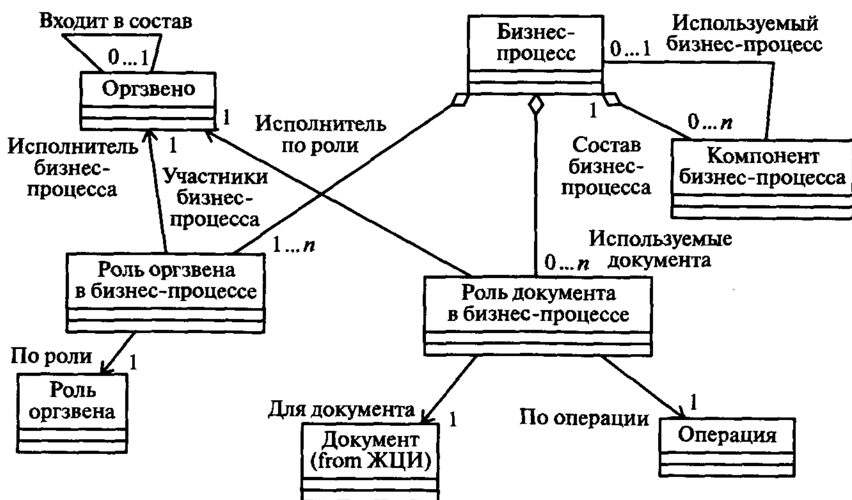


Рис. 7.14. Пример концептуального представления структуры бизнес-процессов предприятия

конкретную операцию в ролевой структуре работы с документами в бизнес-процессах.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Каково назначение каркасов для решения задач управления жизненным циклом изделий? Из чего они состоят?
2. В чем различие моделей спецификации?
3. Объясните зависимости между каркасами для решения задач управления жизненным циклом изделий.
4. Каково назначение каркаса объектного моделирования?
5. В чем заключается проблема классификации такого объекта как Изделие?
6. Как решается проблема моделирования свойств информационных объектов?
7. Каково назначение компонентов модели бизнес-логики на основе кооперации?
8. Каково назначение компонентов модели конструкторской спецификации изделия?
9. Каково назначение компонентов технологической спецификации изделия?
10. Каково назначение компонентов организационно-функциональной модели предприятия?

ЗАКЛЮЧЕНИЕ

Тенденции развития архитектуры ИС связаны с формированием ее ключевых элементов: принципов, стандартов и моделей. Эти основополагающие конструкции взаимосвязаны между собой, и конечной целью их применения является создание абстрактных конструкций, инвариантных к предметной области, которые обеспечивают единство в подходах к проектированию и созданию систем.

Информационные технологии стали важной сферой производственной деятельности с нарастающей динамикой роста, непосредственно влияющей на развитие всей экономики. Перечислим наиболее важные факторы для дальнейшего развития информационной индустрии в сфере использования архитектурных решений:

1) создание полноценного промышленного информационного производства, соединяющего научное (теоретическое), исследовательское и производственное направления;

2) развитие методов, технологий, навыков и инструментальных средств, ориентированных на создание качественных продуктов информационных технологий;

3) комплексная стандартизация как одно из основных направлений промышленного развития информационных технологий. Сформированная международная система стандартов информационных технологий в области производства и образования (объединяет десятки профессиональных организаций) непрерывно развивается;

4) качество и надежность должны стать визитной карточкой информационных продуктов. Основным ориентиром для обеспечения качества должно быть создание условий производства, гарантирующих необходимый уровень качества.

СПИСОК ЛИТЕРАТУРЫ

1. *Буч Г.* UML: Специальный справочник / Г. Буч, Дж. Рамбо, А. Якобсон. — СПб.: Питер, 2002. — 652 с.

2. *Воропаев В. И.* Системное представление управления проектами (СОВНЕТ) / В. И. Воропаев, Г. И. Секлетова. [Электронный ресурс]: http://www.pmuniversity.ru/files/article_1_systematic.pdf.

3. *Гамма Э.* Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм., Р. Джонсон., Дж. Влиссидес. — СПб.: «Питер», 2001. — 368 с.

4. *Гринфилд Д.* Фабрики разработки программ: потоковая сборка типовых приложений, моделирование, структуры и инструменты / Д. Гринфилд, К. Шорт. — М.: Издательский дом «Вильямс»: 2007. — 592 с.

5. *Дрожжин В. В.* Эволюция архитектуры информационных систем / В. В. Дрожжин, Р. Е. Зинченко // Программные продукты и системы. — № 4. — 2010. — С. 59—63.

6. *Кин М.* Основы работы с WebSphere Enterprise Service Bus V6 / М. Кин, Б. Мур, А. Карвальо. — М.: ИБМ, 2007. — 438 с.

7. *Когаловский М. Р.* Перспективные технологии информационных систем / М. Р. Когаловский. — М.: ДМК Пресс; Компания АйТи, 2003. — 288 с.

8. *Когаловский М. Р.* Энциклопедия технологий баз данных / М. Р. Когаловский. — М.: Финансы и статистика, 2002. — 800 с.

9. *Майо Д.* Самоучитель Microsoft Visual Studio / Д. Майо. — СПб.: БХВ-Петербург, 2010. — 464 с.

10. *Оберг Р.* Основы COM+. Основы и программирование / Р. Оберг. — М.: Издательский дом «Вильямс», 2000. — 480 с.

11. *Орфали Р.* Java и CORBA в приложениях клиент-сервер / Р. Орфали, Д. Харки. — М.: Лори, 2000. — 712 с.

12. *Рябов В. А.* Современные веб-технологии / В. А. Рябов, А. И. Несвижский. [Электронный ресурс]: www.intuit.ru/department/internet/mwebtech/ — М.: Интуит, 2010. —475 с.

13. *Советов Б. Я.* Теория информационных процессов и систем / Б. Я. Советов, В. А. Дубенецкий, В. В. Цехановский, О. И. Шеховцов. — М.: Издательский центр «Академия», 2010. — 432 с.

14. Стандарт ISO/IEC 15288 «Системная инженерия — процессы жизненного цикла систем» [Электронный ресурс]: http://www.iso.org/iso/support/faqs.htm.

15. *Стелтинг С.* Применение шаблонов Java. Библиотека профессионала / С. Стелтинг, О. Маасен. — М.: Издательский дом «Вильямс», 2002. — 576 с.

16. *Стивенс У.* Unix: разработка сетевых приложений / У. Стивенс — СПб.: Питер, 2003. — 1088 с.

17. *Таненбаум Э.* Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. — СПб.: Питер, 2003. — 877 с.

18. *Фаулер М.* Шаблоны корпоративных приложений / М. Фаулер. — М.: Издательский дом «Вильямс», 2010. — 544 с.

19. *Хабибуллин И. Ш.* Разработка Web-служб средствами Java / И. Ш. Хабибуллин. — СПб.: БХВ-Петербург, 2003. — 400 с.

20. *Хабибуллин И. Ш.* Создание распределенных приложений на Java 2. / И. Ш. Хабибуллин. — СПб.: БХВ-Петербург, 2002. — 704 с.

21. *Хемраджани А.* Гибкая разработка приложений на Java с помощью Spring, Hibernate и Eclipse / А. Хемраджани. — М.: Издательский дом «Вильямс», 2008. — 352 с.

22. *Хоп Г.* Шаблоны интеграции корпоративных приложений / Г. Хоп, Б. Вульф. — М.: Издательский дом «Вильямс», 2007. — 672 с.

23. *Частиков А. П.* Разработка экспертных систем. Среда CLIPS / А. П. Частиков, Т. А. Гаврилова, Д. Л. Белов. — СПб.: БХВ-Петербург, 2003. — 608 с.

24. *Чеппел Д.* Технологии ActiveX и OLE / Д. Чепппел. — М.: Microsoft Corporation, 1997. — 320 с.

25. AntiPatterns / W. Brown, R. Malveau, H. I. McCormick, T. Mowbray. — N. Y.: John Wiley, 1998. — 156 pp.

26. Apache ServiceMix, the Agile Open Source ESB: [Eelectronic resource] http:// servicemix.apache.org.

27. *Beckner M.* BizTalk 2010 Recipes: A Problem-Solution Approach / M. Beckner. — N. Y.: Apress, 2010. — 608 pp.

28. *Browne P.* JBoss Drools Business Rules / P. Browne. — N. Y. Packt Publishing, 2009. — 304 pp.

29. *Buschmann F.* Pattern-Oriented Software Architecture. Volume 3: Patterns for Resource Management / F. Buschmann, K. Henney, D. Schmidt. — N. Y.: John Wiley, 2004. — 310 pp.

30. *Buschmann F.* Pattern-Oriented Software Architecture. Volume 4: A Pattern Language for Distributed Computing / F. Buschmann, K. Henney, D. Schmidt. — N. Y.: John Wiley, 2007. — 636 pp.

31. *Buschmann F.* Pattern-Oriented Software Architecture. Volume 5: On Patterns And Pattern Languages / F. Buschmann, K. Henney, D. Schmidt. — N. Y.: John Wiley, 2007. — 490 pp.

32. Capability Maturity Model Integration. [Eelectronic resource]: http://www.sei.cmu.edu/cmmi/.

33. *Chappell D.* Enterprise Service Bus / D. Chappell. — Sebastopol: O'Reilly Media, 2004. — 328 pp.

34. *Davies J.* The Definitive Guide to SOA: BEA AquaLogic Service Bus / J. Davies, A. Krishna, D. Schorow D. — N. Y.: Apress, 2007. — 613 pp.

35. *Delia P.* Mule 2: A Developer's Guide to ESB and Integration Platform / P. Delia, A. Borg. — N. Y.: Apress, 2008. — 164 pp.

36. *Duffy D.* Domain Architectures. Models and Architectures for UML Applications / D. Duffy. — Datasim Education BV, Amsterdam, Netherlands, 2004. — 412 pp.

37. *Erl T.* Service-Oriented Architecture: Concepts, Technology, and Design / T. Erl. — N.Y.: Prentice Hall, 2005. — 792 pp

38. *Erl T.* SOA: principles of service design / T. Erl. — N.Y.: Prentice Hall, 2008. — 608 pp.

39. *Friedman-Hill E.* Jess in Action Manning Publications / E. Friedman-Hill. Greenwich: Maning Publication Co, 2003. — 480 c.

40. *von Halle B.* Business Rules Applied: Building Better Systems Using the Business Rules Approach / B. von Halle. — N.Y. Wiley Computer Publishing, 2002. — 592 c.

41. *Ibsen C.* Camel in Action / C. Ibsen, J. Anstey. — Greenwich: Manning Publications, 2011. — 552 pp.

42. ISO 9126 (ГОСТ Р ИСО / МЭК 9126-93)— «Информационная технология. Оценка программного продукта. Характеристики качества и руководство по их применению». Введ. 1994-06-30. — М.: Стандартинформ, 1994. — 12 с.

43. *Iyengar A.* WebSphere Business Integration Primer / A. Iyengar, V. Jessani, M. Chilanti. — N.J.: IBM Press, 2007. — 543 pp.

44. JSR-286 is the Java Portlet specification v2.0http [Eelectronic resource]: http://www.jcp.org/en/jsr/detail?id=286.

45. *Kaisler S.* Software Paradigms / S. Kaisler. — New Jersey: John Wiley & Sons, 2005. — 458 pp.

46. *Khoshafian S.* Service Oriented Enterprises / S. Khoshafian. — N.Y.: Auerbach Publications, 2007. — 464 pp.

47. *Linthicum D. S.* Next Generation Application Integration: From Simple Information to Web Services / D. S. Linthicum. — N.Y.: Addison Wesley, 2003. — 512 pp.

48. MPI: The Message Passing Interface [Electronic resource]: http://parallel.ru/tech/tech_dev/mpi.html.

49. Open ESB: the Open Source for SOA & Integration [Eelectronic resource]: https://open-esb.dev.java.net.

50. *Panda D.* EJB 3 in Action / D. Panda, D. Lane, R. Rahman. — Greenwich: Lane Manning Publications Co., 2007. — 611 pp.

51. Pattern-Oriented Software Architecture. Volume 1: A System of Patterns / F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. — N.Y.: John Wiley, 1996. — 476 pp.

52. Pattern-Oriented Software Architecture. Volume 2: Patterns For Concurrent And Networked Objects / D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. — N.Y.: John Wiley, 2000. — 633 pp.

53. *Polgar J.* Building and managing enterprise-wide portals / J. Polgar, R. Bram, A. Polgar. — London: Idea Group Publishing, 2006. — 304 pp.

54. *Rademakers T.* Open Source ESBs in Action / T. Rademakers, J. Dirksen. — N.Y.: Manning Publications, 2008. — 528 pp.

55. *Szyperski C.* Component Software: Beyond Object-Oriented Programming / C. Szyperski. — N.Y.: Addison Wesley Professional, 2002. — 644 pp.

56. The official website of the United States Department of Defense [Eelectronic resource]: https://www.us.army.mil.

57. The Open Group Architecture Framework [Eelectronic resource]: http://www.opengroup.org/togaf.

58. The Zachman International e-Commerce Site [Electronic resource]: http://www.zachmaninternational.com.

59. *Wang A.* Component-Oriented Programming / A. Wang, K. Qian. — New Jersey: John Wiley & Sons, 2005. — 334 pp.

60. Web Services for Remote Portlets Specification v2.0 OASIS standart [Eelectronic resource]: http://docs.oasis-open.org/wsrp/v2/wsrp-2.0-spec-os-02.html.

61. Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. — N.Y.: Prentice Hall, 2005. — 456 pp.

ОГЛАВЛЕНИЕ

Введение ... 3

Глава 1. Архитектурный подход к информационным системам ... 5

1.1. Основные понятия и определения ... 5
1.2. Характеристика информационной системы как объекта архитектуры ... 8
1.3. Архитектура и проектирование информационных систем ... 22
1.4. Эволюция платформенных архитектур информационных систем ... 30

Глава 2. Архитектурные стили ... 44

2.1. Понятие архитектурного стиля. Классификация архитектурных стилей ... 44
2.2. Потоки данных, вызов с возвратом ... 46
2.3. Независимые компоненты, централизованные данные ... 51
2.4. Виртуальные машины ... 55
2.5. Использование стилей ... 57

Глава 3. Паттерны и фреймворки в архитектуре ИС ... 62

3.1. Паттерны ... 62
3.2. Антипаттерны ... 69
3.3. Фреймворки ... 75
3.4. Примеры фреймворков ... 79

Глава 4. Компонентные технологии реализации информационных систем ... 94

4.1. Понятие компонента. Компонентные технологии ... 94
4.2. Квазикомпонентно-ориентированные технологии ... 98
4.3. Технологии, основанные на объектной модели компонентов COM+, .NET ... 106
4.3.1. Объектная модель компонентов (COM) ... 106
4.3.2. Распределенная объектная модель компонентов (DCOM) ... 115
4.3.3. Технология COM+ ... 115
4.3.4. .NET-компоненты ... 119
4.4. Технология CORBA ... 124
4.5. Технология Enterprise Java Beans ... 134

Глава 5. Сервисно-ориентированные технологии реализации информационных систем 143
5.1. Сервисно-ориентированные архитектуры (СОА) и Web-сервисы ... 143
5.2. Язык XML при работе с Web-сервисами 148
5.3. WSDL-описание 154
5.4. UDDI-реестр 158
5.5. Бизнес-реестр ebXML 161
5.6. Язык WS-Inspection для поиска Web-служб 162
5.7. Спецификации WS-* 163

Глава 6. Интеграция приложений 168
6.1. Общие принципы организации взаимодействий в информационных системах 168
6.2. Интеграция приложений 172
6.3. Системы, ориентированные на работу с сообщениями 175
6.4. Язык описания бизнес-процессов BPEL 179
6.5. Бизнес-правила 193
6.6. Порталы и портлеты 201
6.6.1. Порталы 201
6.6.2. Портлеты 205
6.7. Корпоративные сервисные шины 215
6.7.1. Общие принципы построения 215
6.7.2. Обобщенная архитектурная модель интеграционной подсистемы 221
6.7.3. Существующие решения ESB 224
6.8. Сервисно-ориентированная архитектура и сервисно-ориентированная организация 236

Глава 7. Архитектурные решения разработки приложений 249
7.1. Подходы к архитектурным решениям корпоративных информационных систем 249
7.2. Моделирование структуры классов и их свойств 259
7.3. Поддержки функций приложения 265

Заключение 278

Список литературы 279

Учебное издание

Советов Борис Яковлевич
Водяхо Александр Иванович
Дубенецкий Владислав Алексеевич
Цехановский Владислав Владимирович

Архитектура информационных систем

Учебник

Технический редактор *Н. И. Горбачева*
Компьютерная верстка: *Д. В. Федотов*
Корректоры *А. П. Сизова*, *И. А. Ермакова*

Изд. № 101116069. Подписано в печать 27.04.2012. Формат 60 × 90/16.
Гарнитура «Newton». Бумага офсетная № 1. Печать офсетная. Усл. печ. л. 18,0.
Тираж 1 500 экз. Заказ № 5955.

Издательский центр «Академия». www.academia-moscow.ru

125252, Москва, ул. Зорге, д. 15, корп. 1, пом. 266.
Адрес для корреспонденции: 129085, Москва, пр-т Мира, 101В, стр. 1, а/я 48.
Тел./факс: (495)648-0507, 616-0029.

Санитарно-эпидемиологическое заключение № РОСС RU.AE51.H16067 от 06.03.2012.

Отпечатано с электронных носителей издательства.
ОАО «Тверской полиграфический комбинат», 170024, г. Тверь, пр-т Ленина, 5.
Телефон: (4822) 44-52-03, 44-50-34. Телефон/факс: (4822) 44-42-15.
Home page — www.tverpk.ru Электронная почта (E-mail) — sales@tverpk.ru