

Содержание

| | |
|---|----|
| Предисловие | 17 |
| Введение | 27 |
| Часть I. Обзор | 41 |
| Глава 1. “Расслоение” системы | 43 |
| Развитие модели слоев в корпоративных программных приложениях | 44 |
| Три основных слоя | 46 |
| Где должны функционировать слои | 48 |
| Глава 2. Организация бизнес-логики | 51 |
| Выбор типового решения | 55 |
| Уровень служб | 56 |
| Глава 3. Объектные модели и реляционные базы данных | 59 |
| Архитектурные решения | 59 |
| Функциональные проблемы | 64 |
| Считывание данных | 66 |
| Взаимное отображение объектов и реляционных структур | 67 |
| Отображение связей | 67 |
| Наследование | 71 |
| Реализация отображения | 73 |
| Двойное отображение | 74 |
| Использование метаданных | 75 |
| Соединение с базой данных | 76 |
| Другие проблемы | 78 |
| Дополнительные источники информации | 79 |
| Глава 4. Представление данных в Web | 81 |
| Типовые решения представлений | 84 |
| Типовые решения входных контроллеров | 86 |
| Дополнительные источники информации | 86 |

8 Содержание

| | |
|---|-----|
| Часть II. Типовые решения | 131 |
| Глава 9. Представление бизнес-логики | 133 |
| Сценарий транзакции (Transaction Script) | 133 |
| Принцип действия | 133 |
| Назначение | 135 |
| Задача определения зачетного дохода | 135 |
| Пример: определение зачетного дохода (Java) | 136 |
| Модель предметной области (Domain Model) | 140 |
| Принцип действия | 140 |
| Назначение | 143 |
| Дополнительные источники информации | 143 |
| Пример: определение зачетного дохода (Java) | 144 |
| Модуль таблицы (Table Module) | 148 |
| Принцип действия | 149 |
| Назначение | 151 |
| Пример: определение зачетного дохода (C#) | 152 |
| Слой служб (Service Layer) | 156 |
| Принцип действия | 157 |
| Разновидности “бизнес-логики” | 157 |
| Варианты реализации | 157 |
| Быть или не быть удаленному доступу | 158 |
| Определение необходимых служб и операций | 158 |
| Назначение | 160 |
| Дополнительные источники информации | 160 |
| Пример: определение зачетного дохода (Java) | 161 |
| Глава 10. Архитектурные типовые решения источников данных | 167 |
| Шлюз таблицы данных (Table Data Gateway) | 167 |
| Принцип действия | 167 |
| Назначение | 168 |
| Дополнительные источники информации | 169 |
| Пример: класс PersonGateway (C#) | 170 |
| Пример: использование объектов ADO.NET DataSet (C#) | 172 |
| Шлюз записи данных (Row Data Gateway) | 175 |
| Принцип действия | 175 |
| Назначение | 176 |
| Пример: запись о сотруднике (Java) | 178 |
| Пример: использование диспетчера данных для объекта домена (Java) | 181 |
| Активная запись (Active Record) | 182 |
| Принцип действия | 182 |
| Назначение | 184 |
| Пример: простой класс Person (Java) | 184 |
| Преобразователь данных (Data Mapper) | 187 |
| Принцип действия | 187 |
| Обращение к методам поиска | 190 |

| | |
|--|------------|
| Отображение данных на поля объектов домена | 191 |
| Отображения на основе метаданных | 192 |
| Назначение | 192 |
| Пример: простой преобразователь данных (Java) | 193 |
| Пример: отделение методов поиска (Java) | 198 |
| Пример: создание пустого объекта (Java) | 201 |
| Глава 11. Объектно-реляционные типовые решения, предназначенные для моделирования поведения | 205 |
| Единица работы (Unit of Work) | 205 |
| Принцип действия | 206 |
| Назначение | 211 |
| Пример: регистрация посредством изменяемого объекта (Java) | 212 |
| Коллекция объектов (Identity Map) | 216 |
| Принцип действия | 216 |
| Выбор ключей | 217 |
| Явная или универсальная? | 217 |
| Сколько нужно коллекций? | 217 |
| Куда их поместить? | 218 |
| Назначение | 219 |
| Пример: методы для работы с коллекцией объектов (Java) | 219 |
| Загрузка по требованию (Lazy Load) | 220 |
| Принцип действия | 221 |
| Назначение | 223 |
| Пример: инициализация по требованию (Java) | 224 |
| Пример: виртуальный прокси-объект (Java) | 224 |
| Пример: использование диспетчера значения (Java) | 226 |
| Пример: использование фиктивных объектов (C#) | 227 |
| Глава 12. Объектно-реляционные типовые решения, предназначенные для моделирования структуры | 237 |
| Поле идентификации (Identity Field) | 237 |
| Принцип действия | 237 |
| Выбор ключа | 238 |
| Представление поля идентификации в объекте | 239 |
| Вычисление нового значения ключа | 240 |
| Назначение | 242 |
| Дополнительные источники информации | 243 |
| Пример: числовой ключ (C#) | 243 |
| Пример: использование таблицы ключей (Java) | 244 |
| Пример: использование составного ключа (Java) | 246 |
| Класс ключа | 246 |
| Чтение | 249 |
| Вставка | 252 |
| Обновление и удаление | 256 |
| Отображение внешних ключей (Foreign Key Mapping) | 258 |
| Принцип действия | 258 |
| Назначение | 261 |

10 Содержание

| | |
|--|-----|
| Пример: однозначная ссылка (Java) | 262 |
| Пример: многотабличный поиск (Java) | 265 |
| Пример: коллекция ссылок (C#) | 266 |
| Отображение с помощью таблицы ассоциаций (Association Table Mapping) | 269 |
| Принцип действия | 270 |
| Назначение | 270 |
| Пример: служащие и профессиональные качества (C#) | 271 |
| Пример: использование SQL для непосредственного обращения к базе данных (Java) | 274 |
| Пример: загрузка сведений о нескольких служащих посредством одного запроса (Java) | 278 |
| Отображение зависимых объектов (Dependent Mapping) | 283 |
| Принцип действия | 283 |
| Назначение | 285 |
| Пример: альбомы и композиции (Java) | 285 |
| Внедренное значение (Embedded Value) | 288 |
| Принцип действия | 289 |
| Назначение | 289 |
| Дополнительные источники информации | 290 |
| Пример: простой объект-значение (Java) | 290 |
| Сериализованный крупный объект (Serialized LOB) | 292 |
| Принцип действия | 292 |
| Назначение | 294 |
| Пример: сериализация иерархии отделов в формат XML (Java) | 294 |
| Наследование с одной таблицей (Single Table Inheritance) | 297 |
| Принцип действия | 298 |
| Назначение | 298 |
| Пример: общая таблица игроков (C#) | 299 |
| Загрузка объекта из базы данных | 301 |
| Обновление объекта | 303 |
| Вставка объекта | 303 |
| Удаление объекта | 304 |
| Наследование с таблицами для каждого класса (Class Table Inheritance) | 305 |
| Принцип действия | 305 |
| Назначение | 306 |
| Дополнительные источники информации | 307 |
| Пример: семейство игроков (C#) | 307 |
| Загрузка объекта | 307 |
| Обновление объекта | 310 |
| Вставка объекта | 311 |
| Удаление объекта | 312 |
| Наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance) | 313 |
| Принцип действия | 314 |
| Назначение | 315 |

| | |
|---|-----|
| Пример: конкретные классы игроков (C#) | 316 |
| Загрузка объекта из базы данных | 318 |
| Обновление объекта | 320 |
| Вставка объекта | 320 |
| Удаление объекта | 321 |
| Преобразователи наследования (Inheritance Mappers) | 322 |
| Принцип действия | 323 |
| Назначение | 324 |
| Глава 13. Типовые решения объектно-реляционного отображения с использованием метаданных | 325 |
| Отображение метаданных (Metadata Mapping) | 325 |
| Принцип действия | 326 |
| Назначение | 327 |
| Пример: использование метаданных и метода отражения (Java) | 328 |
| Хранение метаданных | 328 |
| Поиск по идентификатору | 330 |
| Запись в базу данных | 332 |
| Извлечение множества объектов | 334 |
| Объект запроса (Query Object) | 335 |
| Принцип действия | 336 |
| Назначение | 337 |
| Дополнительные источники информации | 337 |
| Пример: простой объект запроса (Java) | 337 |
| Хранилище (Repository) | 341 |
| Принцип действия | 342 |
| Назначение | 343 |
| Дополнительные источники информации | 344 |
| Пример: поиск подчиненных заданного сотрудника (Java) | 344 |
| Пример: выбор стратегий хранилища (Java) | 345 |
| Глава 14. Типовые решения, предназначенные для представления данных в Web | 347 |
| Модель–представление–контроллер (Model View Controller) | 347 |
| Принцип действия | 348 |
| Назначение | 350 |
| Контроллер страниц (Page Controller) | 350 |
| Принцип действия | 351 |
| Назначение | 352 |
| Пример: простое отображение с помощью контроллера-сервлета и представления JSP (Java) | 352 |
| Пример: использование страницы JSP в качестве обработчика запросов (Java) | 355 |
| Пример: обработка запросов страницей сервера с применением механизма разделения кода и представления (C#) | 358 |
| Контроллер запросов (Front Controller) | 362 |
| Принцип действия | 362 |
| Назначение | 364 |

12 Содержание

| | |
|--|------------|
| Дополнительные источники информации | 364 |
| Пример: простое отображение (Java) | 365 |
| Представление по шаблону (Template View) | 368 |
| Принцип действия | 369 |
| Вставка маркеров | 369 |
| Вспомогательный объект | 370 |
| Условное отображение | 370 |
| Итерация | 371 |
| Обработка страницы | 372 |
| Использование сценариев | 372 |
| Назначение | 372 |
| Пример: использование страницы JSP в качестве представления с вынесением контроллера в отдельный объект (Java) | 373 |
| Пример: страница сервера ASP.NET (C#) | 375 |
| Представление с преобразованием (Transform View) | 379 |
| Принцип действия | 379 |
| Назначение | 380 |
| Пример: простое преобразование (Java) | 381 |
| Двухэтапное представление (Two Step View) | 383 |
| Принцип действия | 383 |
| Назначение | 385 |
| Пример: двухэтапное применение XSLT (XSLT) | 390 |
| Пример: страницы JSP и пользовательские дескрипторы (Java) | 393 |
| Контроллер приложения (Application Controller) | 397 |
| Принцип действия | 398 |
| Назначение | 400 |
| Дополнительные источники информации | 400 |
| Пример: модель состояний контроллера приложения (Java) | 400 |
| Глава 15. Типовые решения распределенной обработки данных | 405 |
| Интерфейс удаленного доступа (Remote Facade) | 405 |
| Принцип действия | 406 |
| Интерфейс удаленного доступа и типовое решение интерфейс сеанса (Session Facade) | 409 |
| Слой служб | 409 |
| Назначение | 410 |
| Пример: использование компонента сеанса Java в качестве интерфейса удаленного доступа (Java) | 410 |
| Пример: Web-служба (C#) | 414 |
| Объект переноса данных (Data Transfer Object) | 419 |
| Принцип действия | 419 |
| Сериализация объекта переноса данных | 421 |
| Сборка объекта переноса данных из объектов домена | 423 |
| Назначение | 424 |
| Дополнительные источники информации | 424 |
| Пример: передача информации об альбомах (Java) | 425 |
| Пример: сериализация с использованием XML (Java) | 429 |

| | |
|---|-----|
| Глава 16. Типовые решения для обработки задач автономного параллелизма | 433 |
| Оптимистическая автономная блокировка (Optimistic Offline Lock) | 434 |
| Принцип действия | 435 |
| Назначение | 439 |
| Пример: слой домена с преобразователями данных (Java) | 439 |
| Пессимистическая автономная блокировка (Pessimistic Offline Lock) | 445 |
| Принцип действия | 446 |
| Назначение | 450 |
| Пример: простой диспетчер блокировки (Java) | 450 |
| Блокировка с низкой степенью детализации (Coarse-Grained Lock) | 457 |
| Принцип действия | 457 |
| Назначение | 460 |
| Пример: общая оптимистическая автономная блокировка (Java) | 460 |
| Пример: общая пессимистическая автономная блокировка (Java) | 466 |
| Пример: оптимистическая автономная блокировка корневого элемента (Java) | 467 |
| Неявная блокировка (Implicit Lock) | 468 |
| Принцип действия | 469 |
| Назначение | 470 |
| Пример: неявная пессимистическая автономная блокировка (Java) | 470 |
| Глава 17. Типовые решения для хранения состояния сеанса | 473 |
| Сохранение состояния сеанса на стороне клиента (Client Session State) | 473 |
| Принцип действия | 473 |
| Назначение | 474 |
| Сохранение состояния сеанса на стороне сервера (Server Session State) | 475 |
| Принцип действия | 475 |
| Назначение | 478 |
| Сохранение состояния сеанса в базе данных (Database Session State) | 479 |
| Принцип действия | 479 |
| Назначение | 481 |
| Глава 18. Базовые типовые решения | 483 |
| Шлюз (Gateway) | 483 |
| Принцип действия | 484 |
| Назначение | 484 |
| Пример: создание шлюза к службе отправки сообщений (Java) | 485 |
| Преобразователь (Mapper) | 489 |
| Принцип действия | 490 |
| Назначение | 490 |
| Супертип слоя (Layer Supertype) | 491 |
| Принцип действия | 491 |
| Назначение | 491 |
| Пример: объект домена (Java) | 491 |
| Отделенный интерфейс (Separated Interface) | 492 |
| Принцип действия | 493 |
| Назначение | 494 |

| | |
|---|-----|
| Глава 5. Управление параллельными заданиями | 87 |
| Проблемы параллелизма | 88 |
| Контексты выполнения | 89 |
| Изолированность и устойчивость данных | 91 |
| Стратегии блокирования | 91 |
| Предотвращение возможности несогласованного чтения данных | 93 |
| Разрешение взаимоблокировок | 94 |
| Транзакции | 95 |
| ACID: свойства транзакций | 96 |
| Ресурсы транзакций | 96 |
| Уровни изоляции | 97 |
| Системные транзакции и бизнес-транзакции | 99 |
| Типовые решения задачи обеспечения автономного параллелизма | 101 |
| Параллельные операции и серверы приложений | 102 |
| Дополнительные источники информации | 104 |
| Глава 6. Сеансы и состояния | 105 |
| В чем преимущество отсутствия “состояния” | 105 |
| Состояние сеанса | 107 |
| Способы сохранения состояния сеанса | 108 |
| Глава 7. Стратегии распределенных вычислений | 111 |
| Соблазны модели распределенных объектов | 111 |
| Интерфейсы локального и удаленного вызова | 112 |
| Когда без распределения не обойтись | 114 |
| Сужение границ распределения | 115 |
| Интерфейсы распределения | 116 |
| Глава 8. Общая картина | 119 |
| Предметная область | 120 |
| Источник данных | 121 |
| Источник данных для сценария транзакции | 121 |
| Источник данных для модуля таблицы | 122 |
| Источник данных для модели предметной области | 122 |
| Слой представления | 123 |
| Платформы и инструменты | 124 |
| Java и J2EE | 124 |
| .NET | 125 |
| Хранимые процедуры | 126 |
| Web-службы | 126 |
| Другие модели слоев | 127 |

14 Содержание

| | |
|---|------------|
| Реестр (Registry) | 495 |
| Принцип действия | 495 |
| Назначение | 497 |
| Пример: реестр с единственным экземпляром (Java) | 498 |
| Пример: реестр, уникальный в пределах потока (Java) | 499 |
| Объект-значение (Value Object) | 500 |
| Принцип действия | 501 |
| Назначение | 502 |
| Совпадение названий | 502 |
| Деньги (Money) | 502 |
| Принцип действия | 503 |
| Назначение | 506 |
| Пример: класс Money (Java) | 506 |
| Частный случай (Special Case) | 511 |
| Принцип действия | 512 |
| Назначение | 512 |
| Дополнительные источники информации | 512 |
| Пример: объект NullEmployee (C#) | 513 |
| Дополнительный модуль (Plugin) | 514 |
| Принцип действия | 514 |
| Назначение | 515 |
| Пример: генератор идентификаторов (Java) | 516 |
| Фиктивная служба (Service Stub) | 519 |
| Принцип действия | 519 |
| Назначение | 520 |
| Пример: служба определения величины налога (Java) | 521 |
| Множество записей (Record Set) | 523 |
| Принцип действия | 524 |
| Явный интерфейс | 524 |
| Назначение | 526 |
| Список основных источников информации | 527 |
| Предметный указатель | 532 |

ЧАСТЬ I

Обзор

Глава 1

“Расслоение” системы

Концепция *слоев (layers)* — одна из общеупотребительных моделей, используемых разработчиками программного обеспечения для разделения сложных систем на более простые части. В архитектурах компьютерных систем, например, различают слои кода на языке программирования, функций операционной системы, драйверов устройств, наборов инструкций центрального процессора и внутренней логики чипов. В среде сетевого взаимодействия протокол FTP работает на основе протокола TCP, который, в свою очередь, функционирует “поверх” протокола IP, расположенного “над” протоколом Ethernet.

Описывая систему в терминах архитектурных слоев, удобно воспринимать составляющие ее подсистемы в виде “слоеного пирога”. Слой более высокого уровня пользуется услугами, предоставляемыми нижележащим слоем, но тот не “осведомлен” о наличии соседнего верхнего слоя. Более того, обычно каждый промежуточный слой “скрывает” нижний слой от верхнего: например, слой 4 пользуется услугами слоя 3, который обращается к слою 2, но слой 4 не знает о существовании слоя 2. (Не в каждой архитектуре слои настолько “непроницаемы”, но в большинстве случаев дело обстоит именно так.)

Расчленение системы на слои предоставляет целый ряд преимуществ.

- Отдельный слой можно воспринимать как единое самодостаточное целое, не особенно заботясь о наличии других слоев (скажем, для создание службы FTP необходимо знать протокол TCP, но не тонкости Ethernet).
- Можно выбирать альтернативную реализацию базовых слоев (приложения FTP способны работать без каких-либо изменений в среде Ethernet, по соединению PPP или в любой другой среде передачи информации).
- Зависимость между слоями можно свести к минимуму. Так, при смене среды передачи информации (при условии сохранения функциональности слоя IP) служба FTP будет продолжать работать как ни в чем не бывало.
- Каждый слой является удачным кандидатом на стандартизацию (например, TCP и IP — стандарты, определяющие особенности функционирования соответствующих слоев системы сетевых коммуникаций).
- Созданный слой может служить основой для нескольких различных слоев более высокого уровня (протоколы TCP/IP используются приложениями FTP, telnet, SSH и HTTP). В противном случае для каждого протокола высокого уровня пришлось бы изобретать собственный протокол низкого уровня.

Схема расслоения обладает и определенными недостатками.

- Слои способны удачно инкапсулировать многое, но не все: модификация одного слоя подчас связана с необходимостью внесения каскадных изменений в остальные слои. Классический пример из области корпоративных программных приложений: поле, добавленное в таблицу базы данных, подлежит воспроизведению в графическом интерфейсе и должно найти соответствующее отображение в каждом промежуточном слое.
- Наличие избыточных слоев нередко снижает производительность системы. При переходе от слоя к слою моделируемые сущности обычно подвергаются преобразованиям из одного представления в другое. Несмотря на это, инкапсуляция нижележащих функций зачастую позволяет достичь весьма существенного преимущества. Например, оптимизация слоя транзакций обычно приводит к повышению производительности всех вышележащих слоев.

Однако самое трудное при использовании архитектурных слоев — это определение содержимого и границ ответственности каждого слоя.

Развитие модели слоев в корпоративных программных приложениях

По молодости лет мне не довелось пообщаться с ранними версиями систем пакетной обработки данных, но, как мне кажется, тогда люди не слишком задумывались о каких-то там “слоях”. Писалась программа, которая манипулировала некими файлами (ISAM, VSAM и т.п.), и этим, собственно говоря, функции приложения исчерпывались.

Понятие слоя приобрело очевидную значимость в середине 1990-х годов с появлением систем *клиент/сервер* (*client/server*). Это были системы с двумя слоями: клиент нес ответственность за отображение пользовательского интерфейса и выполнение кода приложения, а роль сервера обычно поручалась СУБД. Клиентские приложения создавались с помощью таких инструментальных средств, как Visual Basic, PowerBuilder и Delphi, предоставлявших в распоряжение разработчика все необходимое, включая экранные компоненты, обслуживающие интерфейс SQL: для конструирования окна было достаточно перетащить на рабочую область необходимые управляющие элементы, настроить параметры доступа к базе данных и подключиться к ней, используя таблицы свойств.

Если задачи сводились к простым операциям по отображению информации из базы данных и ее незначительному обновлению, системы клиент/сервер действовали безотказно. Проблемы возникли с усложнением логики предметной области — бизнес-правил, алгоритмов вычислений, условий проверок и т.д. Прежде все эти обязанности возлагались на код клиента и находили отражение в содержимом интерфейсных экранов. Чем сложнее становилась логика, тем более неуклюжим и трудным для восприятия делался код. Воспроизведение элементов логики на экранах приводило к дублированию кода, и тогда при необходимости внести простейшее изменение приходилось “прочесывать” всю программу в поисках одинаковых фрагментов.

Одной из альтернатив было описание логики в тексте хранимых процедур, размещаемых в базе данных. Языки хранимых процедур, однако, отличались ограниченными возможностями структуризации, что вновь негативно сказывалось на качестве кода. Помимо

того, многие отдали предпочтение реляционным системам баз данных, поскольку используемый в них стандартизованный язык SQL открывал возможности безболезненного перехода от одной СУБД к другой. Хотя воспользовались ими на практике только единицы, мысль о возможной смене поставщика СУБД, не связанной со сколько-нибудь ощутимыми затратами, согревала всех. А наличие жесткой зависимости языков хранимых процедур от конкретных версий систем фактически разрушало эти надежды.

По мере роста популярности систем клиент/сервер набирала силу и парадигма объектно-ориентированного программирования, давшая сообществу ответ на sacramentalный вопрос о том, куда “девать” бизнес-логику: перейти к системной архитектуре с *тремя* слоями, в которой слой *представления* отводится пользовательскому интерфейсу, слой *предметной области* предназначен для описания бизнес-логики, а третий слой представляет *источник данных*. В этом случае удалось бы разнести интерфейс и логику, поместив последнюю на отдельный уровень, где она может быть структурирована с помощью соответствующих объектов.

Несмотря на предпринятые усилия, движение под знаменем объектной ориентации в направлении трехуровневой архитектуры было еще слишком робким и неуверенным. Многие проекты оказывались чрезмерно простыми, что отнюдь не вызывало у программистов желания покинуть наезженную колею систем клиент/сервер и связать себя новыми обязательствами. Помимо того, средства разработки приложений клиент/сервер с трудом поддерживали трехуровневую модель вычислений либо не предоставляли подобных инструментов вовсе.

Радикальный сдвиг произошел с появлением Web. Всем внезапно захотелось иметь системы клиент/сервер, где в роли клиента выступал бы Web-обозреватель. Если, однако, вся бизнес-логика приложения сосредоточивалась в коде *толстого* клиента, при переходе к Web-интерфейсу приходилось пересматривать ее полностью. А в удачно спроектированной трехуровневой системе достаточно было просто заменить уровень представления, не затрагивая слой предметной области. Позже, с появлением Java, все увидели объектно-ориентированный язык, претендующий на всеобщее признание. Появившиеся инструментальные средства конструирования Web-страниц были в меньшей степени связаны с SQL и потому более подходили для реализации третьего уровня.

При обсуждении вопросов расслоения программных систем нередко путают понятия *слоя (layer)* и *уровня*, или *яруса (tier)*. Часто их употребляют как синонимы, но в большинстве случаев термин *уровень* трактуют, подразумевая *физическое* разделение. Поэтому системы клиент/сервер обычно описывают как двухуровневые (в общем случае “клиент” действительно отделен от сервера физически): клиент — это приложение для настольной машины, а сервер — процесс, выполняемый сетевым компьютером-сервером. Я применяю термин *слой*, чтобы подчеркнуть, что слои вовсе *не* обязательно должны располагаться на разных машинах. Отдельный слой бизнес-логики может функционировать как на персональном компьютере “рядом” с клиентским слоем интерфейса, так и на сервере базы данных. В подобных ситуациях речь идет о двух узлах сети, но о трех слоях или уровнях. Если база данных локальна, все три слоя могут соседствовать и на одном компьютере, но даже в этом случае они должны сохранять свой суверенитет.

Три основных слоя

В этой книге внимание акцентируется на архитектуре с тремя основными слоями: *представление (presentation)*, *домен (предметная область, бизнес-логика) (domain)* и *источник данных (data source)*. В табл. 1.1 приведено их краткое описание (названия заимствованы из [9]).

Таблица 1.1. Основные слои

| Слой | Функции |
|-----------------|--|
| Представление | Предоставление услуг, отображение данных, обработка событий пользовательского интерфейса (щелчков кнопками мыши и нажатий клавиш), обслуживание запросов HTTP, поддержка функций командной строки и API пакетного выполнения |
| Домен | Бизнес-логика приложения |
| Источник данных | Обращение к базе данных, обмен сообщениями, управление транзакциями и т.д. |

Слой *представления* охватывает все, что имеет отношение к общению пользователя с системой. Он может быть настолько простым, как командная строка или текстовое меню, но сегодня пользователю, вероятнее всего, придется иметь дело с графическим интерфейсом, оформленным в стиле толстого клиента (Windows, Swing и т.п.) или основанным на HTML. К главным функциям слоя представления относятся отображение информации и интерпретация вводимых пользователем команд с преобразованием их в соответствующие операции в контексте домена (бизнес-логики) и источника данных.

Источник данных — это подмножество функций, обеспечивающих взаимодействие со сторонними системами, которые выполняют задания в интересах приложения. Код этой категории несет ответственность за мониторинг транзакций, управление другими приложениями, обмен сообщениями и т.д. Для большинства корпоративных приложений основная часть логики источника данных сосредоточена в коде СУБД.

Логика *домена (бизнес-логика или логика предметной области)* описывает основные функции приложения, предназначенные для достижения поставленной перед ним цели. К таким функциям относятся вычисления на основе вводимых и хранимых данных, проверка всех элементов данных и обработка команд, поступающих от слоя представления, а также передача информации слою источника данных.

Иногда слои организуют таким образом, чтобы бизнес-логика полностью скрывала источник данных от представления. Чаше, однако, код представления может обращаться к источнику данных непосредственно. Хотя такой вариант менее безупречен с теоретической точки зрения, в практическом отношении он нередко более удобен и целесообразен: код представления может интерпретировать команду пользователя, активизировать функции источника данных для извлечения подходящих порций информации из базы данных, обратиться к средствам бизнес-логики для анализа этой информации и осуществления необходимых расчетов и только затем отобразить соответствующую картинку на экране.

Часто в рамках приложения предусматривают несколько вариантов реализации каждой из трех категорий логики. Например, приложение, ориентированное на использование как интерфейсных средств толстого клиента, так и командной строки, может (и, вероятно, должно) быть оснащено двумя соответствующими версиями логики представления. С другой стороны, различным базам данных могут отвечать многочисленные слои источников данных. На отдельные “пакеты” может быть поделен даже слой бизнес-логики (скажем, в ситуации, когда алгоритмы расчетов зависят от типа источника данных).

До сих пор предполагалось обязательное наличие пользователя. Возникает закономерный вопрос: что произойдет, если в управлении программным приложением человек участия не принимает (примерами могут служить и новейшие Web-службы, и традиционный процесс пакетной обработки)? В этом случае в роли пользователя приложения выступает сторонняя клиентская программа и становится очевидным сходство между слоями представления и источника данных: оба они задают связь приложения с внешним миром. Именно этот вариант логики лежит в основе типового решения **шестигранная архитектура (Hexagonal Architecture)** Алистера Коуберна (Alistair Cockburn) [39], которое трактует любую систему как ядро, окруженное интерфейсами к внешним системам. В **шестигранной архитектуре**, где все, что находится снаружи, — это не что иное, как интерфейсы к внешним субъектам, исповедуется симметричный подход к проблеме, в отличие от асимметричной схемы расслоения, которой придерживаюсь я.

Эта асимметрия, однако, кажется мне полезной, поскольку, как я полагаю, следует различать интерфейс, предлагаемый в виде службы другим, и сторонние службы, которыми пользуетесь вы сами. Собственно говоря, в этом и состоит реальное различие между слоями представления и источника данных. Представление — это внешний интерфейс к службе, который приложение открывает стороннему потребителю: либо человеку-оператору, либо программе. Источник данных — это интерфейс к функциям, которые предлагаются приложению внешней системой. Я нахожу очевидные выгоды в том, что интерфейсы трактуются как неодинаковые, поскольку это различие заставляет по-особому воспринимать каждую из служб.

Хотя три основных слоя — представление, бизнес-логика и источник данных — можно обнаружить в любом корпоративном приложении, способ их разделения зависит от степени сложности этого приложения. Простой сценарий извлечения порции информации из базы данных и отображения ее в контексте Web-страницы можно описать одной процедурой. Но я все равно попытался бы выделить в нем три слоя — пусть даже, как в этом случае, распределив функции каждого слоя по разным подпрограммам. При усложнении приложения это дало бы возможность разнести код слоев по отдельным классам, а позже разбить множество классов на пакеты. Форма расслоения может быть произвольной, но в любом корпоративном приложении слои *должны* быть идентифицированы.

Помимо необходимости разделения на слои, существует правило, касающееся взаимоотношения слоев: зависимость бизнес-логики и источника данных от уровня представления не допускается. Другими словами, в тексте приложения не должно быть вызовов функций представления *из* кода бизнес-логики или источника данных. Правило позволяет упростить возможность адаптации слоя представления или замены его альтернативным вариантом с сохранением основы приложения. Связь между бизнес-логикой и источником данных, однако, не столь однозначна и во многом определяется выбором типовых решений для архитектуры источника данных.

Самым сложным в работе над бизнес-логикой является, вероятно, выбор того, *что* именно и *как* следует относить к тому или иному слою. Мне нравится один неформальный тест. Вообразите, что в программу добавляется принципиально отличный слой, например интерфейс командной строки для Web-приложения. Если существует некий набор функций, которые придется продублировать для осуществления задуманного, значит, здесь логика домена “перетекает” в слой представления. Можно сформулировать тест иначе: нужно ли повторять логику при необходимости замены реляционной базы данных XML-файлом?

Хорошим примером ситуации является система, о которой мне когда-то рассказали. Представьте, что приложение отображает список выделенных красным цветом названий товаров, объемы продаж которых возросли более чем на 10% в сравнении с уровнем прошлого месяца. Допустим, программист разместил соответствующую логику непосредственно в слое представления, решив здесь же сопоставлять уровни продаж текущего и прошлого месяца и изменять цвет, если разность превышает заданный порог.

Проблема заключается в том, что в слой представления вводится несвойственная ему логика предметной области. Чтобы надлежащим образом разделить слои, нужен метод бизнес-логики, отображающий факт превышения уровня продаж определенного продукта на заданную величину. Метод должен осуществить сравнение уровней продаж по двум месяцам и вернуть значение булевого типа. В коде слоя представления достаточно вызвать этот метод и, руководствуясь полученным результатом, принять решение об изменении цвета отображения. В этом случае процесс разбивается на две части: выявление факта, который может служить основанием для изменения цвета, и собственно изменение.

Впрочем, мне не хотелось бы выглядеть сухим доктринером. Рецензируя эту книгу, Алан Найт (Alan Knight) как-то признался, что он “разрывался между тем, считать ли передачу подобной функции в ведение пользовательского интерфейса первым шагом на скользкой дорожке, ведущей напрямик в преисподнюю, либо вполне разумным компромиссным решением, с которым не согласился бы только отъявленный буквоед”. Причина, которая беспокоит нас обоих, состоит в том, что на самом деле верны *оба* вывода!

Где должны функционировать слои

На протяжении всей книги речь идет о логических слоях, т.е. о расчленении системы на отдельные части. Подобное разделение полезно даже тогда, когда все слои функционируют на одной машине. Впрочем, существуют ситуации, где различия в поведении системы могут быть обусловлены принципами ее физической организации.

В большинстве случаев существует только два варианта размещения и выполнения компонентов корпоративных приложений — на персональном компьютере и на сервере.

Зачастую самым простым является функционирование кода всех слоев системы на сервере. Это становится возможным, например, при использовании HTML-интерфейса, воспроизводимого Web-обозревателем. Основным преимуществом сосредоточения всех частей приложения в одном месте является то, что при этом максимально упрощаются процедуры исправления ошибок и обновления версий. В этом случае не приходится беспокоиться о внесении соответствующих изменений на всех компьютерах, об их совместности с другими приложениями и синхронизации с серверными компонентами.

Общие аргументы в пользу размещения каких-либо слоев на компьютере клиента состоят в повышении *быстроты реагирования* (*responsiveness*) приложения и в обеспечении возможности локальной работы. Чтобы код сервера смог отреагировать на действия, предпринимаемые пользователем на клиентской машине, требуется определенное время. А если пользователю необходимо быстро опробовать несколько вариантов и немедленно увидеть результат, продолжительность сетевого обмена становится серьезным препятствием. Помимо того, приложению требуется сетевое соединение как таковое. В частности, размышляя над этими строками, я находился в десяти километрах от ближайшего пункта, где можно было бы подключиться к сети, но хотелось бы, чтобы сетевой доступ обеспечивался везде. Может быть, в обозримом будущем так и случится, но что делать жителям какой-нибудь тьмутаракани, которые не желают ждать, пока кто-то из операторов беспроводной связи удосужится обеспечить “покрытие” их Богом забытого селения? А поддержка возможностей локального функционирования выдвигает особые требования, но боюсь, что они выбиваются из контекста этой книги.

Приняв к сведению все приведенные соображения, можно исследовать альтернативы, рассматривая слой за слоем. Слой источника данных лучше всегда располагать на сервере. Исключение составляет случай, когда функции сервера дублируются в коде “очень толстого” клиента для обеспечения средств локального функционирования системы. При этом предполагается, что изменения, вносимые в отдельные источники данных на клиентской машине и на сервере, подлежат синхронизации посредством механизма репликации. Однако, как уже упоминалось выше, обсуждение подобных вопросов придется отложить до лучших времен или передать инициативу другому автору.

Решение о том, где должен функционировать слой представления, большей частью зависит от предпочтений в выборе типа пользовательского интерфейса. Применение интерфейса толстого клиента автоматически влечет за собой необходимость размещения слоя представления на клиентской машине. Использование Web-интерфейса означает, что логика представления сосредоточена на сервере. Существуют и исключения, например удаленное управление клиентским программным обеспечением (таким, как X-сервер в UNIX) с запуском Web-сервера на настольном компьютере, но они редки.

Если речь идет о создании системы типа “поставщик–потребитель” (“business to customer” — B2C), у вас просто нет выбора. К серверу может подключиться любой, и вы вряд ли будете мириться с потерей посетителя только из-за того, что он использует какое-то экзотическое программное или аппаратное обеспечение. Поэтому целесообразно все функции сконцентрировать на сервере, а клиенту передавать материал в формате HTML, полностью готовый для воспроизведения с помощью Web-обозревателя. Подобное архитектурное решение ограничено в том, что реализация самой незначительной логики пользовательского интерфейса требует обращения к серверу, а это не может не сказаться на скорости реагирования приложения. Уменьшить зависимость от сервера можно за счет применения фрагментов кода на языках сценариев Web-обозревателя (подобных JavaScript) и загружаемых апплетов, но подобные меры снижают уровень совместимости обозревателей и вызывают другие проблемы. Чем более “чист” код HTML, тем проще жизнь.

Вряд ли ваша жизнь будет простой даже в том случае, если каждый из настольных компьютеров вашей компании настроен, как утверждает начальник отдела информационных технологий, “максимально тщательно”. Необходимость поддержки клиентского программного обеспечения в актуальном состоянии и требование исключить даже малую

вероятность его несовместимости с другими программами — это серьезные проблемы, которые проявляются и в тривиальных ситуациях.

Основной повод для применения интерфейсов толстого клиента — сложность задач и невозможность создания полноценных полезных приложений иной архитектуры. Однако популярность Web-интерфейсов неуклонно растет, а потребность в использовании толстых клиентов, напротив, снижается. Могу сказать одно: пользуйтесь Web-интерфейсами, если можете, и обращайтесь к средствам толстого клиента, если без них никак не обойтись.

А как быть с кодом бизнес-логики? Его можно активизировать или целиком на сервере, или полностью в контексте клиентской части, или используя смешанный стиль. И вновь вариант “все на сервере” наиболее привлекателен с точки зрения удобства сопровождения системы. Передача каких-либо бизнес-функций клиенту может быть обусловлена только, скажем, необходимостью повышения скорости реагирования интерфейса системы или потребностью в средствах поддержки локального функционирования.

Если в рамках клиента необходимо выполнять какие-либо функции логики предметной области, прежде всего уместно рассмотреть возможность поручения клиенту *всех* таких функций. Подобный вариант очень похож на выбор интерфейса толстого клиента. Запуск Web-сервера на клиентской машине ненамного повысит скорость реагирования приложения, хотя даст возможность использовать его в локальном режиме. Где бы ни находился код бизнес-логики, его следует сохранять в отдельных модулях, не связанных со слоем представления, используя одно из типовых решений — **сценарий транзакции (Transaction Script, 133)** или **модель предметной области (Domain Model, 140)**. Передача клиенту всего кода бизнес-логики сопровождается — и это уже отмечалось — усложнением процедур обновления системы.

Расщепление множества бизнес-функций между сервером и клиентом выглядит как наихудшее решение, поскольку в общем случае затрудняет идентификацию того или иного фрагмента логики. Основная причина, побуждающая применять подобную архитектуру, может состоять в том, что клиенту необходимо владеть только какой-то частью бизнес-логики. Главное — изолировать эту порцию кода в отдельном модуле, не зависящем от других частей системы. Это даст возможность активизировать код и на компьютере клиента, и на сервере, если такая потребность возникнет позже. Такой подход, разумеется, требует дополнительных усилий, но они оправданны.

После выбора узлов обработки необходимо попытаться обеспечить выполнение всего кода, относящегося к каждому отдельному узлу, в рамках единого процесса, функционирующего либо на одном узле, либо в пределах кластера из нескольких узлов. Не стоит делить слои по разрозненным процессам, если в этом нет насущной необходимости. В противном случае вам придется иметь дело с решениями типа **интерфейса удаленного доступа (Remote Facade, 405)** и **объекта переноса данных (Data Transfer Object, 419)**, а это чревато потерей производительности и повышением сложности.

Важно помнить, что подобные вещи относятся к числу тех, которые Дженс Коулдвей (Jens Coldewey) метко окрестила *катализаторами сложности (complexity boosters)*: это распределенная обработка, многопоточные вычисления, сочетание радикально различных концепций (например, “объектной ориентации” и “реляционной модели”), межплатформенное взаимодействие и обеспечение предельно высокого уровня быстродействия. Решение любой из названных задач сопряжено с большими затратами. Конечно, иногда приходится их нести, но это должно рассматриваться как исключение, а не правило.