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 службами, 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 сакраментальный 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. Чаще, 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 преставления 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 возвратить значение булева 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 прямиком 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 совместимости 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 раздельные 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 быстроте реагирования приложения. Уменьшить зависимость от сервера можно за счет применения фрагментов кода на языках сценариев Web-обозревателя (подобных JavaScript) и загружаемых аплетов, 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 быстроты 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 быстроту 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.