Введение

Создание компьютерных систем (может быть, вы этого еще не знаете) — весьма непростое дело. По мере увеличения их сложности трудоемкость процессов конструирования соответствующего программного обеспечения возрастает согласно экспоненциальному закону. Как и в любой профессии, прогресс в программировании достигается только путем обучения, причем как на ошибках, так и на удачах — своих и чужих. Это издание представляет собой учебное пособие, которое поможет вам усвоить информацию и передать полученные знания другим значительно быстрее и эффективнее, чем это удавалось мне самому.

Ниже обозначен контекст книги и освещаются базовые понятия и идеи, на которых зиждется дальнейшее изложение материала.

Архитектура

Одно из странных свойств, присущих индустрии программного обеспечения, связано с тем, что какой-либо термин может иметь множество противоречивых толкований. Вероятно, наиболее многострадальным оказалось понятие *архитектура (architecture)*. Мне кажется, оно принадлежит к числу тех нарочито эффектных слов, которые употребляются преимущественно для обозначения чего-то, считающегося значительным и серьезным. (Нет, я слишком прагматичен, чтобы цинично пользоваться этим фактом, стараясь привлечь внимание читающей публики. :–))

Термин “архитектура” пытаются трактовать все, кому не лень, и всяк на свой лад. Впрочем, можно назвать два общих варианта. Первый связан с разделением системы на наиболее крупные составные части; во втором случае имеются в виду некие конструктивные решения, которые после их принятия с трудом поддаются изменению. Также растет понимание того, что существует более одного способа описания архитектуры и степень важности каждого из них меняется в продолжение жизненного цикла системы.

Время от времени Ралф Джонсон (Ralph Johnson), участвуя в списке рассылки, отправлял туда замечательные сообщения, одно из которых, касающееся проблем архитектуры, совпало с периодом завершения мною чернового варианта книги. В этом сообщении он подтвердил мое мнение о том, что архитектура — весьма субъективное понятие. В лучшем случае оно отображает общую точку зрения команды разработчиков на результаты проектирования системы. Обычно это согласие в вопросе идентификации главных компонентов системы и способов их взаимодействия, а также выбор таких решений,

которые интерпретируются как основополагающие и не подлежащие изменению в будущем. Если позже оказывается, что нечто изменить легче, чем казалось вначале, это "нечто" легко исключается из "архитектурной" категории.

В этой книге представлено мое в*и*дение основных компонентов корпоративного приложения и решений, которые должны приниматься на ранних фазах его жизни. Мне по душе трактовка системы в виде набора архитектурных *слоев (layers)* (подробнее о слоях — в главе 1, "«Расслоение» системы"). Поэтому в книге предлагаются ответы на вопросы, как осуществить декомпозицию системы по слоям и как обеспечить надлежащее взаимодействие слоев между собой. В большинстве корпоративных приложений прослеживается та или иная форма архитектурного "расслоения", но в некоторых ситуациях большее значение могут приобретать другие подходы, связанные, например, с организацией *каналов (pipes)* или *фильтров (filters)*. Однако мы сконцетрируем внимание на архитектуре слоев как на наиболее плодотворной структурной модели.

Одни типовые решения, рассмотренные ниже, можно определенно считать "архитектурными" в том смысле, что они представляют "значимые" составные части приложения и/или "основополагающие" аспекты функционирования этих частей. Другие относятся к вопросам реализации. Но я не собираюсь классифицировать, что в большей, а что в меньшей степени "архитектурно", чтобы не навязывать вам свое мнение.

Корпоративные приложения

Созданием программного обеспечения занимается множество людей. Но говорить о программном обеспечении в целом — значит, не сказать ничего, поскольку разновидностей программных приложений чрезвычайно много и каждая ситуация выдвигает особые проблемы, отличающиеся собственным уровнем сложности. Наглядным примером могут служить дискуссии с коллегами, работающими в сфере телекоммуникаций. С определенной точки зрения "мои" корпоративные приложения намного проще, нежели телекоммуникационное программное обеспечение, — мне не приходится решать чрезвычайно сложные проблемы обеспечения многопоточного функционирования и тесной интеграции аппаратных и программных компонентов. Но во всем остальном мои задачи существенно сложнее. Корпоративные приложения чаще всего имеют дело с изощренными данными большого объема и бизнес-правилами, логика которых иногда просто противоречит здравому смыслу. Хотя некоторые приемы и решения более-менее универсальны, большинство из них адекватны только в контексте определенных ситуаций.

На протяжении всей профессиональной карьеры я занимался преимущественно корпоративными приложениями, и это, разумеется, обусловило выбор типовых решений, представленных на страницах книги. (В числе близких аналогов понятия "приложение" можно назвать термин "информационная система"; читатели постарше, вероятно, вспомнят выражение "обработка данных".) Но что именно подразумевает понятие *корпоративное приложение (enterprise application)*? Точное определение сформулировать трудно, но дать смысловое толкование вполне возможно.

Начнем с примеров. К числу корпоративных приложений относятся, скажем, бухгалтерский учет, ведение медицинских карт пациентов, экономическое прогнозирование, анализ кредитной истории клиентов банка, страхование, внешнеэкономические торговые

операции и т.п. Корпоративными приложениями *не* являются средства обработки текста, регулирования расхода топлива в автомобильном двигателе, управления лифтами и оборудованием телефонных станций, автоматического контроля химических процессов, а также операционные системы, компиляторы, игры и т.д.

Корпоративные приложения обычно подразумевают необходимость долговременного (иногда в течение десятилетий) *хранения данных*. Данные зачастую способны "пережить" несколько поколений прикладных программ, предназначенных для их обработки, аппаратных средств, операционных систем и компиляторов. В продолжение этого срока структура данных может подвергаться многочисленным изменениям в целях сохранения новых порций информации без какого-либо воздействия на старые. Даже в тех случаях, когда компания осуществляет революционные изменения в парке оборудования и номенклатуре программных приложений, данные не уничтожаются, а переносятся в новую среду.

Данных, с которыми имеет дело корпоративное приложение, как правило, бывает *много*: даже скромная система способна манипулировать несколькими гигабайтами информации, организованной в виде десятков миллионов записей; и задача манипуляции этими данными вырастает в одну из основных функций приложения. В старых системах информация хранилась в виде индексированных файловых структур, подобных разработанным компанией IBM VSAM и ISAM. Сейчас для этого применяются системы управления базами данных (СУБД), большей частью реляционные. Проектирование таких систем и их сопровождение превратились в отдельные специализированные дисциплины.

Множество пользователей обращаются к данным *параллельно*. Как правило, их количество не превышает сотни, но для систем, размещенных в среде Web, этот показатель возрастает на несколько порядков. Как гарантировать возможность одновременного доступа к базе данных для всех, кто имеет на это право? Проблема остается даже в том случае, если речь идет всего о двух пользователях: они должны манипулировать одним элементом данных только такими способами, которые исключают вероятность возникновения ошибок. Большинство обязанностей по управлению параллельными заданиями принимает на себя диспетчер транзакций из состава СУБД, но полностью избавить прикладные системы от подобных забот программистам чаще всего не удается.

Если объемы данных столь велики, в приложении должно быть предусмотрено и множество различных вариантов *окон экранного интерфейса*. Вполне обычна ситуация, когда программа содержит несколько сотен окон. Одни пользователи работают с корпоративным приложением регулярно, другие обращаются к нему эпизодически, но полагаться на их техническую осведомленность в любом случае нельзя. Поэтому данные должны допускать возможность представления в самых разных формах, удобных для пользователей всех категорий. Фокусируя внимание на вопросах взаимодействия пользователей с приложением, нельзя упускать из виду и тот факт, что многие системы характеризуются высокой степенью пакетной обработки данных.

Корпоративные приложения редко существуют в изоляции. Обычно они требуют *интеграции* с другими системами, построенными в разное время и с применением различных технологий (файлы данных COBOL, CORBA, системы обмена сообщениями). Время от времени корпорации стараются провести интеграцию собственных подсистем с применением некоторой универсальной технологии. Поскольку обычно такой работе конца-краю не видно, все сводится к применению нескольких различных методов интеграции.

Ситуация усугубляется, если интеграции подлежит информация из совершенно разнородных источников.

Даже в том случае, если компания унифицирует способ интеграции своих подсистем, проблемы на этом не заканчиваются: новым камнем преткновения выступает различие в ведении бизнес-процессов и *концептуальный диссонанс* в представлении данных. В одном подразделении компании под клиентом подразумевают лицо, с которым заключено действующее соглашение; в другом к клиентам относят и тех, с кем приходилось работать прежде; в третьем учитывают только товарные сделки, но не оказанные услуги и т.п. На первый взгляд все это не кажется слишком серьезным, но если каждое из полей в сотнях типов записей обладает какой-то особой семантикой, легкие облака над вашей головой в один момент могут превратиться в грозовые тучи (достаточно представить, что тот единственный на всю компанию человек, который знал все нюансы, вдруг уволился). (Для того чтобы увидеть полную картину происходящего, следует учесть также тот факт, что данные подвержены постоянным изменениям, вносимым, как правило, без какого-либо предупреждения.)

Не стоит забывать и о том, *что* принято обозначать расплывчатым термином *бизнес-логика*. Я нахожу его забавным, поскольку могу припомнить только несколько вещей, менее логичных, нежели так называемая бизнес-логика. При создании операционной системы, например, без строгой логики не обойтись. Но бизнес-правила, которые нам сообщают, следует принимать такими, какие они есть, поскольку без серьезного политического вмешательства их не преодолеть. Мы вынуждены иметь дело со случайными наборами странных условий, которые сочетаются между собой самым непредсказуемым образом. Определенно известно только одно: вся эта мешанина еще и изменяется во времени. Разумеется, тому есть какие-то причины. Так, например, выгодный клиент может выговорить для себя особые условия оплаты кредита, отвечающие срокам поступления средств на его расчетный счет. И пару тысяч таких вот частных случаев способны сделать бизнес-логику совершенно *нелогичной*, а соответствующее программное приложение — запутанным и не поддающимся восприятию с позиций здравого смысла.

Некоторые интерпретируют понятие “корпоративное приложение” как синоним термина “большая система”. Однако важно понимать, что не все подобные приложения на самом деле “велики”, хотя их вклад в деятельность “корпорации” может быть довольно весомым. Многие считают, что “малые” системы причиняют меньше беспокойства, и до определенной степени это справедливо. Крах небольшого приложения обычно менее ощутим, нежели сбой крупной системы. Впрочем, я думаю, что пренебрегать совокупным влиянием многих мелких систем непозволительно. Если вы в состоянии предпринять какие-либо действия, которые помогут улучшить функционирование небольших приложений, общий результат может оказаться весьма существенным, так как нередко значимость системы далеко превосходит ее “размеры”. И еще. При любой возможности старайтесь превратить “большой” проект в “малый” за счет упрощения принимаемой архитектуры и ограничения множества реализуемых функций.

Типы корпоративных приложений

При обсуждении способов проектирования корпоративных приложений и вариантов выбора типовых решений важно понимать, что двух одинаковых приложений просто не существует и различия в проблемах обусловливают необходимость применения различных средств достижения поставленных целей. Меня захлестывает волна протеста, когда я слышу советы делать что-либо "только так и не иначе". По моему убеждению, наиболее любопытная и дерзкая часть проекта, как правило, связана с анализом пространства альтернатив и взвешенным выбором в пользу одной из них. Рассмотрим для примера три довольно типичных варианта корпоративных приложений.

Первый — электронная коммерческая система типа "поставщик–потребитель" ("business to customer" — B2C); к ней обращаются с помощью Web-обозревателей, находят нужные товары, вводят необходимую информацию и осуществляют покупки. Подобная система предназначена для обслуживания большого количества пользователей одновременно, поэтому проектное решение должно быть не только эффективным по критерию использования ресурсов, но и масштабируемым: все, что требуется для повышения пропускной способности такой системы, — это приобретение дополнительного аппаратного обеспечения. Бизнес-логика подобного приложения довольно прямолинейна: прием заказа, незамысловатые финансовые операции и отправка уведомления о доставке. Необходимо, чтобы каждый мог быстро и легко обратиться к системе, поэтому Web-интерфейс должен быть предельно простым и доступным для воспроизведения с помощью максимально широкого диапазона обозревателей. Информационный источник включает базу данных для хранения заказов и, возможно, некий механизм обмена данными с системой складского учета для получения информации о наличии товаров и их отгрузке.

Сопоставьте рассмотренную систему с программой, автоматизирующей учет соглашений имущественного найма. В некоторых аспектах последняя намного проще, нежели приложение электронной коммерции B2C, поскольку круг ее пользователей, работающих одновременно, существенно *уже* — скажем, не более сотни. В чем она сложнее, так это в бизнес-логике. Составление ежемесячных счетов за аренду, обработка различных событий (таких, как преждевременный возврат имущества и просроченный платеж), проверка вводимой информации при заключении договора — все это достаточно трудоемкие функции. В индустрии лизинга успех во многом определяется выбором одного из множества вариантов, незначительно отличающихся от классических примеров сделок, которые заключались в прошлом. И бизнес-логика этой предметной области весьма сложна, поскольку правила игры слишком свободны.

Подобная система отличается сложностью и в отношении пользовательского интерфейса. Зачастую требования к интерфейсу таковы, что их нельзя удовлетворить только средствами HTML; приходится пользоваться интерфейсными средствами, предоставляемыми более традиционной моделью "толстого" клиента. Усложнение процедур взаимодействия пользователя с программой вынуждает применять и более изощренные варианты транзакций: например, оформление договора аренды может продолжаться несколько часов, и все это время пользователь выполняет одну логическую транзакцию. Схема базы данных также заметно расширяется и может включать несколько сотен таблиц и соединений с внешними пакетами, предназначенными для оценки стоимости активов и цены аренды.

В качестве третьего примера рассмотрим простейшую систему учета расходов для небольшой компании, обладающую самой незатейливой логикой. Пользователи системы (их всего несколько) могут обращаться к ней с помощью стандартизованного Web-интерфейса. Источником информации является база данных с двумя-тремя таблицами. Однако даже к такой тривиальной задаче нельзя относиться легкомысленно. Вероятно, систему потребуется сконструировать очень быстро, в то же время принимая во внимание возможности ее роста за счет будущего включения модулей финансового анализа и налогообложения, генерации отчетной документации, взаимодействия с другими приложениями и т.п. Попытки применения тех же архитектур, которые приемлемы в двух предыдущих случаях, в данной ситуации чреваты существенным замедлением темпов работ. Если программа обеспечивает получение тех или иных преимуществ (а такими должны быть все корпоративные приложения), задержка с ее внедрением в эксплуатацию означает прямые финансовые потери. Поэтому отнюдь не хотелось бы принимать решения, которые воспрепятствуют развитию системы в дальнейшем. Однако, если оснастить систему дополнительными службами с прицелом на будущее, но сделать это неправильно, новый уровень сложности как раз и может затруднить ее эволюцию, приостановить процесс внедрения и отсрочить получение преимуществ. Несмотря на "незначительность" каждого отдельного приложения такого класса, все они, рассматриваемые в совокупности, способны оказать серьезное положительное (или отрицательное) влияние на показатели деятельности "корпорации".

Реализация каждой из трех упомянутых систем сопряжена с трудностями, но трудности эти в каждом случае специфичны. Как следствие, нам не удастся предложить единую архитектуру, приемлемую во всех ситуациях. Выбирая архитектуру и варианты проектирования, следует принимать во внимание особенности конкретной системы. Вот почему на страницах книги вы не найдете универсальных рецептов, пригодных на все случаи жизни. Напротив, наличие многих типовых решений не избавляет вас от необходимости выбора из нескольких альтернатив. Даже если вы отдадите предпочтение какому-либо решению, вам, возможно, придется модифицировать его, чтобы приспособить к реальным условиям. Нельзя работать над проектом корпоративного приложения бездумно. Все, на что способна какая бы то ни было книга, — это дать некую исходную информацию, оттолкнувшись от которой вы можете начать мыслить самостоятельно.

Все сказанное применимо и к выбору инструментальных средств программирования. Приступая к работе, следует — думаю, это очевидно — максимально сузить круг инструментов. Однако не стоит забывать, что разные средства проявляют свои лучшие качества в разных ситуациях. Избегайте использования инструментов, предназначенных для других целей, иначе они не только не принесут пользы, но и причинят вред.

Производительность

Многие архитектурные решения напрямую связаны с аспектами *производительности (performance)* системы. Чтобы говорить о производительности, я предпочитаю увидеть работающую систему, измерить ее характеристики и, учитывая полученные результаты, применить строго определенные процедуры оптимизации. Однако некоторые архитектурные решения определяют параметры производительности таким образом, что

устранение возможных проблем средствами оптимизации затрудняется. Именно поэтому к принятию таких решений всегда стоит подходить с большой ответственностью.

В книгах, подобных этой, трудно толковать вопросы производительности, поскольку любой совет не может приниматься как незыблемый факт до тех пор, пока не будет испытан в конкретных условиях. Я слишком часто сталкивался с ситуациями, когда вариант проектирования принимался или отвергался по причинам, связанным с производительностью, а затем замеры параметров осуществлялись в реальной среде и полностью опровергали прежние выводы.

На страницах книги я приведу всего несколько рекомендаций по повышению производительности приложений, включая и такую, которая касается минимизации количества удаленных вызовов. Однако какими бы неоспоримыми все они ни казались, вы обязаны всегда проверять их на практике. Помимо того, в некоторых примерах кода я пожертвовал соображениями эффективности выполнения в угоду удобочитаемости. Выполняя оптимизацию, не забывайте проверять параметры производительности до и после, иначе вы добьетесь только того, что код станет более трудным для восприятия.

Существует одно важное обстоятельство: значительное изменение конфигурации приложения способно перечеркнуть все ранее установленные факты, касающиеся вопросов производительности. Поэтому, обновив версию виртуальной машины, СУБД, аппаратного обеспечения или любого другого компонента системы, вы обязаны заново провести оптимизацию и убедиться, что принятые меры возымели действие. Нередко бывает и так, что приемы оптимизации, с успехом применявшиеся в прошлом, в новых условиях вызывают обратный эффект.

Еще одна проблема любой дискуссии по вопросам производительности состоит в том, что многие термины употребляются и интерпретируются непоследовательно и неверно. Наиболее характерный пример — понятие, описывающее *способность приложения к масштабированию (scalability)*: тут можно назвать не менее пяти-шести различных толкований. Ниже рассмотрены термины, применяемые в этой книге.

Время отклика (response time) — промежуток времени, который требуется системе, чтобы обработать запрос извне, подобный щелчку на кнопке графического интерфейса или вызову функции API сервера.

Быстрота реагирования (responsiveness) — скорость подтверждения запроса (не путать с временем отклика — скоростью обработки). Эта характеристика во многих случаях весьма важна, поскольку интерактивная система, пусть даже обладающая нормальным временем отклика, но не отличающаяся высокой быстротой реагирования, всегда вызывает справедливые нарекания пользователей. Если, прежде чем принять очередной запрос, система должна полностью завершить обработку текущего, параметры времени отклика и быстроты реагирования, по сути, совпадают. Если же система способна подтвердить получение запроса раньше, ее быстрота реагирования выше. Например, применение динамического индикатора состояния процесса копирования повышает быстроту реагирования экранного интерфейса, хотя никак не сказывается на значении времени отклика.

Время задержки (latency) — минимальный интервал времени до получения какого-либо отклика (даже если от системы более ничего не требуется). Параметр приобретает особую важность в распределенных системах. Если я “прикажу” программе ничего не делать и сообщить о том, когда именно она закончит это “ничегонеделание”, на персональном компьютере ответ будет получен практически мгновенно. Если же программа

выполняется на удаленной машине, придется подождать, вероятно, не менее нескольких секунд, пока запрос и ответ проследуют по цепочке сетевых соединений. Снизить время задержки разработчику прикладной программы не под силу. Фактор задержки — главная причина, побуждающая минимизировать количество удаленных вызовов.

Пропускная способность (throughput) — количество данных (операций), передаваемых (выполняемых) в единицу времени. Если, например, тестируется процедура копирования файла, пропускная способность может измеряться числом байтов в секунду. В корпоративных приложениях обычной мерой производительности служит число транзакций в секунду (transactions per second — tps), но есть одна проблема — транзакции различаются по степени сложности. Для конкретной системы необходимо рассматривать смесь "типовых" транзакций.

В контексте рассмотренных терминов под *производительностью* можно понимать один из двух параметров — *время отклика* или *пропускную способность*, в частности тот, который в большей степени отвечает природе ситуации. Иногда бывает трудно судить о производительности, если, например, использование некоторого решения повышает пропускную способность, одновременно увеличивая время отклика. С точки зрения пользователя, значение быстроты реагирования может оказаться более важным, нежели время отклика, так что улучшение быстроты реагирования ценой потери пропускной способности или возрастания времени отклика вполне способно повысить производительность.

Загрузка (load) — значение, определяющее степень "давления" на систему и измеряемое, скажем, количеством одновременно подключенных пользователей. Параметр загрузки обычно служит контекстом для представления других функциональных характеристик, подобных времени отклика. Так, нередко можно слышать выражения наподобие следующего: "время отклика на запрос составляет 0,5 секунды для 10 пользователей и 2 секунды для 20 пользователей".

Чувствительность к загрузке (load sensitivity) — выражение, задающее зависимость времени отклика от загрузки. Предположим, что система A обладает временем отклика, равным 0,5 секунды для 10–20 пользователей, а система B — временем отклика в 0,2 секунды для 10 пользователей и 2 секунды для 20 пользователей. Это дает основание утверждать, что система A обладает меньшей чувствительностью к загрузке, нежели система B. Можно воспользоваться и термином *ухудшение (degradation)*, чтобы подчеркнуть факт меньшей устойчивости параметров системы B.

Эффективность (efficiency) — удельная производительность в пересчете на одну единицу ресурса. Например, система с двумя процессорами, способная выполнить 30 tps, более эффективна по сравнению с системой, оснащенной четырьмя аналогичными процессорами и обладающей продуктивностью в 40 tps.

Мощность (capacity) — наибольшее значение пропускной способности или загрузки. Это может быть как абсолютный максимум, так и некоторое число, при котором величина производительности все еще превосходит заданный приемлемый порог.

Способность к масштабированию (scalability) — свойство, характеризующее поведение системы при добавлении ресурсов (обычно аппаратных). Масштабируемой принято считать систему, производительность которой возрастает пропорционально объему приобщенных ресурсов (скажем, вдвое при удвоении количества серверов). *Вертикальное масштабирование (vertical scalability, scaling up)* — это увеличение мощности отдельного сервера (например, за счет увеличения объема оперативной памяти). *Горизонтальное*

масштабирование (horizontal scalability, scaling out) — это наращивание потенциала системы путем добавления новых серверов.

Следует отметить, что существует проблема: выбор проектного решения оказывает не одинаковое влияние на факторы производительности. Рассмотрим для примера две программные системы, работающие на однотипных серверах: первая (назовем ее Swordfish) обладает мощностью в 20 tps, а вторая, Camel, — в 40 tps. Какая система более производительна? Какая из них в большей мере способна к масштабированию? На основании имеющейся информации мы не можем ответить на вопрос, касающийся возможностей масштабирования. Единственное, что понятно, — Camel более эффективна в конфигурации с одним сервером. Добавим еще один сервер и допустим, что производительность Swordfish теперь исчисляется величиной 35 tps, а продуктивность Camel возрастает до 50 tps. Мощность Camel все еще выше, но параметры масштабирования Swordfish выглядят предпочтительнее. Продолжив пополнение систем серверами, мы обнаружим, что производительность Swordfish всякий раз возрастает на 15 tps, в то время как аналогичный показатель Camel — только на 10 tps. Полученные данные позволяют прийти к заключению, что Swordfish лучше поддается горизонтальному масштабированию, хотя Camel демонстрирует бо́льшую эффективность при количестве серверов, меньшем пяти.

Проектируя корпоративную систему, часто следует уделять больше внимания обеспечению средств масштабирования, а не наращиванию мощности или повышению эффективности. Производительность масштабируемой системы всегда удастся увеличить, если такая потребность действительно возникнет. Помимо того, добиться возможности масштабирования проще. Нередко проектировщикам и программистам приходится изворачиваться, чтобы увеличить мощность системы для конкретной аппаратной конфигурации, хотя, вероятно, было бы дешевле приобрести дополнительное оборудование. Если, скажем, система Camel стоит дороже, чем Swordfish, и разница в стоимости покрывает цену нескольких серверов, то Swordfish обойдется дешевле даже в том случае, когда необходимый уровень производительности составляет всего 40 tps. Сейчас модно сетовать на возрастающие требования к аппаратному обеспечению, непременно сопровождающие выпуск очередных версий популярных программ, и я присоединяю свой голос к хору возмущенных потребителей, когда при установке последней версии Word мне велят обновить конфигурацию компьютера. Но использовать более новое оборудование зачастую просто выгоднее, нежели заставлять программу “крутиться” на устаревшей технике. Если корпоративное приложение поддается масштабированию, добавить несколько серверов дешевле, чем приобрести услуги нескольких программистов.

Типовые решения

Концепция *типовых решений (patterns)* известна уже давно, и мне не хотелось бы выступать здесь с очередным очерком истории ее развития. Однако я не собираюсь упускать удачную возможность рассказать о собственной точке зрения на предмет.

Какого бы то ни было общеупотребительного определения типового решения не существует. Вероятно, лучше всего начать с цитаты из книги Кристофера Александера (Christopher Alexander), вдохновившей не одно поколение энтузиастов: “Каждое типовое решение описывает некую повторяющуюся проблему и ключ к ее разгадке, причем таким

образом, что вы можете пользоваться этим ключом многократно, ни разу не придя к одному и тому же результату" [1]. К. Александер — архитектор, и говорит он о сооружениях, но данное им определение, по моему мнению, прекрасно подходит и для типовых решений в программировании. Основа типового решения — подход, достаточно общий и эффективный для того, чтобы обеспечить преодоление периодически возникающих проблем определенной природы. Типовое решение можно воспринимать и в виде некоторой рекомендации: искусство создания типовых решений состоит в вычленении и кристаллизации таких относительно независимых рекомендаций, которые допускают возможность более или менее обособленного применения.

Типовые решения уходят своими корнями в практику. Чтобы распознать их, необходимо присмотреться к тому, что и как делают другие, изучить полученные результаты и выявить их главную составляющую. Это нелегко, но коль скоро хорошие типовые решения найдены, они обретают большую ценность. Для меня эта ценность в данный момент проявляется в том, что дает основание написать книгу, которую будут использовать как руководство к действию. Вам не обязательно читать ее (или любое другое аналогичное издание) от корки до корки, чтобы убедиться в полезности предлагаемых рекомендаций. Достаточно краткого знакомства, которое позволит почувствовать смысл проблем и соответствующих решений. От вас не требуется досконально знать все детали, кроме тех, которые помогут подыскать в книге подходящее типовое решение. И лишь потом вам, возможно, понадобится изучить его более подробно.

Как только вы ощутили потребность в типовом решении, следует подумать, как применить его в конкретных обстоятельствах. Основная особенность типовых решений состоит в том, что ими нельзя пользоваться слепо; вот почему многие инструментальные оболочки, основанные на типовых решениях, и получаемые с их помощью результаты так убоги. Мне кажется, что типовые решения можно сравнить с полуфабрикатом: чтобы получить удовольствие от еды, вам придется довершить приготовление блюда по собственному рецепту. Используя типовое решение, я "поджариваю" его всякий раз по-другому и наслаждаюсь отменным и своеобразным вкусом.

Каждое типовое решение относительно независимо от других, но говорить об их полной взаимной изолированности нельзя. Зачастую одно решение непосредственно приводит к другому либо может применяться только в контексте некоего базового решения. Так, например, решение **наследование с таблицами для каждого класса (Class Table Inheritance, 305)** обычно используется совместно с решением **модель предметной области (Domain Model, 140)**. Границы между решениями не отличаются четкостью, но я предпринял все возможное для того, чтобы представить каждое решение в самодостаточной форме. Поэтому, если кто-то предлагает "применить решение **единица работы (Unit of Work, 205)**", будет довольно проследовать к соответствующему разделу книги, не читая всего остального.

Если у вас есть опыт проектирования корпоративных приложений, многие типовые решения, о которых речь пойдет ниже, вам, вероятно, уже знакомы. Надеюсь, вы не будете разочарованы (ваше возможное недоумение я уже пытался предупредить в предисловии). Типовые решения — не научные открытия; они в большой мере представляют собой обобщения результатов, накопленных в соответствующей области. Поэтому более уместно говорить о "поиске" типового решения, а не о его "изобретении". Моя роль состоит в обнаружении общих подходов, выделении главного и оформлении полученных выводов. Ценность типового решения для проектировщика заключается не в новизне

идеи; главное — это помощь в описании и сообщении этой идеи. Если собеседники осведомлены о том, *что* именно представляет собой **интерфейс удаленного доступа (Remote Facade, 405)**, одному из них достаточно произнести фразу "этот класс является **интерфейсом удаленного доступа**", чтобы расставить все точки над *i*. А новичку вы сможете подсказать, что, дескать, "для этой задачи больше подойдет **объект переноса данных (Data Transfer Object, 419)**", и он, обратившись к странице 646 настоящей книги, найдет все необходимое. Классификация типовых решений способствует созданию словаря проектировщика, а потому выбору их названий следует уделять должное внимание.

В главе 18, "Базовые типовые решения", упомянут ряд базовых типовых решений универсального характера; они используются при описании всех остальных решений, ориентированных на корпоративные приложения.

Структура типовых решений

Каждому автору приходится выбирать форму представления типовых решений. Некоторые принимают за основу подходы, которые изложены в классических книгах, посвященных типовым решениям (например, [1, 20, 34]). Другие изобретают что-то свое. Я долго не мог составить твердое мнение о том, в чем же состоит превосходство того или иного варианта над остальными. С одной стороны, меня не могло удовлетворить нечто столь же краткое, как форма из [20]; с другой — описание каждого решения должно было бы уместиться в пределах небольшого раздела книги. Ниже описан компромиссный результат, к которому я пришел после длительных размышлений.

Первое — название решения. Выбор подходящего имени довольно важен, поскольку одна из целей, преследуемых при использовании типовых решений, состоит в создании словаря, который облегчает взаимодействие проектировщиков в ходе выполнения проекта. Например, если я сообщу, что мой Web-сервер реализован на основе **контроллера запросов (Front Controller, 362)** и **представления с преобразованием (Transform View, 379)**, и эти типовые решения будут знакомы и вам, у вас наверняка сложится однозначное представление об архитектуре приложения.

За именем следуют аннотация и эскиз. Аннотация состоит из одного-двух предложений и содержит краткое описание типового решения; эскиз служит визуальной иллюстрацией решения и часто (хотя не всегда) представляет собой UML-диаграмму. Аннотация и эскиз предназначены для создания наглядного "образа" решения и облегчают его запоминание. Если у вас уже есть решение (в том смысле, что оно вам известно, хотя, возможно, вы не знаете его названия), аннотация и эскиз — это, пожалуй, все, с помощью чего можно его быстро отыскать.

Далее вкратце обозначается проблема, которая стимулировала появление типового решения. Если таких проблем несколько, выбирается более характерная. Затем приводятся два обязательных раздела.

В разделе *Принцип действия* освещаются особенности реализации решения. Изложение ведется в стиле, в наименьшей мере зависящем от специфики какой бы то ни было конкретной платформы. Там, где подобная информация все-таки целесообразна, соответствующий текст выделен с помощью отступов, сразу привлекающих внимание читателя. Чтобы упростить восприятие материала, в некоторых случаях здесь приводятся дополнительные UML-диаграммы.

Раздел *Назначение* содержит сведения о том, при каких обстоятельствах следует применять типовое решение, а также о его сравнительных преимуществах и недостатках.

Многие из решений, упомянутых в этой книге, взаимозаменяемы: таковыми являются, скажем, **контроллер страниц (Page Controller, 350)** и **контроллер запросов (Front Controller, 362)**. Часто одинаково пригодными оказываются несколько решений, поэтому, обнаружив одно из них, я спрашиваю себя, в каких случаях им не стоило бы пользоваться. Ответ на этот вопрос нередко подводит меня к выбору альтернативного решения.

В разделе *Дополнительные источники информации* приводятся ссылки на отдельные библиографические источники, имеющие отношение к обсуждаемой теме. Внимание ограничивается тем материалом, который, как мне кажется, способен оказать наибольшую помощь в освоении особенностей типового решения. Из рассмотрения исключены источники, не содержащие ничего нового, работы, с которыми я не знаком лично, публикации в редких или экзотических изданиях, а также нестабильные и сомнительные гиперссылки.

В конце раздела, посвященного тому или иному решению, по мере необходимости могут быть приведены *простые примеры* использования, проиллюстрированные фрагментами кода на языках Java и/или C#. Выбор языков программирования (это уже отмечалось в предисловии) обусловлен вероятными предпочтениями большинства читателей. Очень важно понимать, что любой пример *не есть* типовое решение. При реализации решения оно *не* будет выглядеть точно так же, как пример, поэтому примеры не следует трактовать как некую разновидность макросов. Я намеренно пытался упрощать их настолько, насколько это было возможно, так что соответствующие типовые решения предстанут перед вами в такой прозрачной форме, какую я только мог вообразить. Стремясь к простоте, я силился продемонстрировать в примерах сущность типовых решений и старался избегать деталей, которые могли бы отвлечь внимание от главного. Все частные проблемы опущены, но имейте в виду, что они могут быстро всплыть в условиях конкретной конфигурации системы. Вот почему типовые решения всегда следует испытывать на практике.

Вместо рассмотрения одного или нескольких сквозных примеров я решил прибегнуть к простым независимым примерам. Последние легче воспринимать, хотя не всегда понятно, как использовать совместно. Сквозной пример демонстрирует проблемы в их связи, но не удобен для быстрого знакомства с природой конкретного решения. Я не исключаю возможности компромисса, но мне найти его не удалось.

Код примеров написан так, чтобы обеспечить простоту восприятия. В результате некоторые вещи остались, что называется, “за кадром” (в частности, так произошло с обработкой ошибок — вопросом, недостаточно хорошо проработанным мною для того, чтобы предложить здесь какое-либо типовое решение). Каждый пример обставлен слишком большим количеством упрощающих условий, и поэтому я посчитал нецелесообразным обеспечить возможность загрузки образцов кода с моего Web-сайта.

Ограничения предлагаемых в книге типовых решений

Как упоминалось в предисловии, предлагаемую коллекцию типовых решений *нельзя* трактовать как исчерпывающее руководство по разработке корпоративных программных приложений. Я рассматривал бы эту книгу не с позиций полноты, а с точки зрения полезности. Предмет слишком серьезен для одной головы, тем более для одной публикации.

Представленные здесь сведения касаются всех типовых решений в области корпоративных приложений, известных мне на данный момент. Я не собираюсь утверждать, что полностью осведомлен обо всех их разновидностях и взаимозависимостях. Книга

отражает уровень моего сегодняшнего опыта с учетом тех знаний, которые были приобретены в процессе ее написания. Но мысль не стоит на месте, и опыт, разумеется, будет пополняться и после выхода книги в свет. Программирование — бесконечный процесс, и в нем нет предела совершенству.

Собираясь воспользоваться типовыми решениями, не забывайте, что они только отправная точка, а не пункт назначения. Ни один автор не в состоянии предвидеть всего многообразия форм и вариантов программных проектов. Я написал эту книгу, чтобы помочь вам сдвинуться с места, так что извлекайте уроки. Помните, что любое типовое решение — это начало трудного пути, и только вам решать, стоит ли брать на себя ответственность (и не лишать себя удовольствия) пройти его до конца.