



БИБЛИОТЕКА ПРОГРАММИСТА



БИБЛИОТЕКА ПРОГРАММИСТА

Сергей Тарасов

ДЕФРАГМЕНТАЦИЯ МОЗГА СОФТОСТРОЕНИЕ ИЗНУТРИ



Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Ростов-на-Дону • Екатеринбург • Самара • Новосибирск
Киев • Харьков • Минск

2013

ББК 32.973.2-018

УДК 004.41

T19

Тарасов С.

T19 Дефрагментация мозга. Софтостроение изнутри. — СПб.: Питер, 2013. — 224 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-00606-4

Эта книга для тех, кто давно связан с разработкой программного обеспечения. Или для тех, кто еще думает выбрать программирование своей профессией. Или для тех, кто просто привык думать и размышлять о происходящем в мире информационных технологий.

Не секрет, что основная масса софтостроения сосредоточена в секторе так называемой корпоративной разработки: от комплексных информационных систем предприятия до отдельных приложений. Поэтому немалая часть сюжетов касается именно Enterprise Programming.

Из текста вы вряд ли узнаете, как правильно склеивать многоэтажные постройки из готовых компонентов в гетерогенной среде, проектировать интерфейсы, синхронизировать процессы или писать эффективные запросы к базам данных. Подобные темы будут лишь фоном для рассказа о софтостроительной «кухне». При определенной доле любопытства вы сможете убедиться, что новое — это хорошо забытое старое, узнать, как устроены некоторые сложные системы, когда следует применять разные технологии, почему специалистам в информатике надо особенно тщательно фильтровать поступающую из множества источников информацию, и многое другое, что вы, возможно, еще не знали или уже знаете, но с другой стороны.

В книге мне хотелось показать наш софтостроительный мир разработки корпоративных информационных систем не с парадного фасада описаний программных сред, подходов и технологий, а изнутри. Насколько это получилось — судить читателю.

ББК 32.973.2-018

УДК 004.41

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-496-00606-4

© ООО Издательство «Питер», 2013

Содержание

О нашей профессии.	10
Очень краткий экскурс.	10
Специализация	11
Кто такой ведущий инженер, или Как это было	12
Иерархия подразделений.	13
Иерархия должностей	13
Уровни принятия проектных решений	14
Функциональные специализации (роли).	15
Метаморфозы	16
О красоте	17
6 миллионов на раздел пирога.	18
Круговорот	20
Масштабы и последствия.	23
Профориентация	25
Начинающим соискателям	27
Про CV	28
Про мотивацию	30
Изгибы судьбы при поиске работы.	31

Технологии 35

Можно ли конструировать программы как аппаратуру?	36
Безысходное программирование	39
Эволюция аппаратуры и скорость разработки	40
Диалог о производительности	43
О карманных монстрах.	44
ASP.NET и браузеры	46
Апплеты, Flash и Silverlight	50
ООП – неизменно стабильный результат	54
ORM, или объектно-реляционный проектор	61
Соккрытие базы данных, или Как скрестить ежа с ужом	61
Как обычно используют ORM	64
Триггер как идеальная концепция для NHibernate	66
ORM на софтостроительной площадке	68
Эмпирика	71
ВЦКП в облаках	72
SaaS и манипуляции терминами	74
От CORBA к SOA	75
Прогресс неотвратим	84
.NET	86
Office 2007	87
SQL Server	88
Vista	90
О материальном	90

Проектирование и процессы 92

Краткий словарь для начинающего проектировщика	92
Слоистость и уровни	93
Концептуальное устройство	94

Логическое устройство	95
Физическое устройство	95
Уровни	97
Совмещение	98
Многозвенная архитектура.	98
История нескольких <code>#ifdef</code>	100
Начало	101
<code>#ifdef NWSQL</code> (1991–92 год)	102
<code>#ifdef BTSQL</code> (1992–93 год)	103
NDL, или Java в миниатюре (1993–94 год)	104
Закат Novell.	106
<code>#ifdef Windows</code>	107
<code>#ifdef MSSQL</code>	108
Постскрипtum	108
Ultima-S – КИС из коробки	109
Нешаблонное мышление	122
Думать головой	126
Обобщение	126
Про сборку мусора и агрегацию.	128
Журнал хозяйственных операций.	131
UML и птолемеевские системы.	135
Когда старая школа молода	140
«Оптисток», или распределённый анализ данных	143
Архитектура сокрытия проблем	148
Code revision, или Коза кричала	151
Наживулька или гибкость?	154
Тесты и практика продуктового софтостроения	163
Говорящие изменения в MSF и выключатель	165
Приключения с TFS	166
Программная фабрика: дайте мне модель, и я сдвину Землю	170

Лампа, полная джиннов.	174
Слой хранения (СУБД).	181
Слой домена (NHibernate)	183
Слой веб-служб и интерфейсов доступа (ServiceStack)	186
Программа-клиент.	192
Остановиться и оглянуться	193
Cherchez le bug, или Программирование по-французски	194
Хаос наступает внезапно	195
Что-то с памятью моей стало.	198
Три дня в IBM	200
Хорошо там, где нас нет	203
 О технических книгах	 205
Дефрагментация мозгов	205
Простые правила чтения специальной литературы.	208
Литература и программное обеспечение	209
 Вместо послесловия, или Краткое изложение «Оснований»	 212
 Литература	 217

К читателю

Эта книга для тех, кто давно связан с разработкой программного обеспечения. Или для тех, кто ещё только думает о выборе программирования в качестве своей профессии. Или для тех, кто просто привык думать и размышлять о происходящем в мире информационных технологий.

Не секрет, что основная масса софтверостроения сосредоточена в секторе так называемой корпоративной разработки: от комплексных информационных систем предприятия до отдельных приложений. Поэтому немалая часть сюжетов касается именно Enterprise Programming.

В процессе чтения книги вы вряд ли узнаете, как правильно склеивать многоэтажные постройки из готовых компонентов в гетерогенной среде, проектировать интерфейсы, синхронизировать процессы или писать эффективные запросы к базам данных. Подобные темы будут лишь фоном для рассказа о софтверостроительной «кухне». При определённой доле любопытства вы убедитесь, что новое — это хорошо забытое старое, узнаете, как устроены некоторые сложные системы, когда следует применять разные технологии, почему специалистам в информатике надо особенно тщательно фильтровать поступающую информацию и многое другое, что вы, возможно, ещё не знали или знали, но с другой стороны.

В книге мне хотелось показать наш софтверостроительный мир разработки корпоративных информационных систем не с парадного фасада описаний программных сред, подходов и технологий, а изнутри. Насколько это получилось — судить читателю.

О нашей профессии

У каждого дела запах особый:
В булочной пахнет тестом и сдобой...

Д. Родари. «Чем пахнут ремёсла?»

Очень краткий экскурс

Более полувека прошло с момента появления первых электронно-вычислительных машин — монстров на базе реле и электронных ламп, занимавших целые здания. Современное уместающееся на ладони устройство во много раз превосходит по вычислительной мощности любого из своих не столь уж дальних предков.

Несколько раз сменилась элементная база, отгремела микропроцессорная революция миниатюризации, изменились технологии, реальностью стали общедоступные вычислительные центры коллективного пользования и глобальная сеть. Неизменной осталась лишь суть профессии программиста. По-прежнему, программист — это человек, способный заставить компьютер решать поставленное перед ним множество задач.

Если первые программисты были сильно ограничены в средствах и привязаны к аппаратному обеспечению — «железу», к конкретной ЭВМ¹, то современные располагают огромным арсеналом инструментов и технологий, в большинстве

¹ Электронная Вычислительная Машина, вдруг кто забыл, как назывались компьютеры в русском языке.

случаев позволяющих разработчику не принимать во внимание особенности устройства тех компьютеров, на которых его программа будет выполняться.

С одной стороны, работа, с технологической точки зрения, облегчилась, автоматизировался весь процесс, от написания кода до сборки и компоновки. С другой стороны, требования к желаемому состоянию «задача решена» стали не просто более сложными, но и во многих случаях более неопределёнными. Появилась огромная масса проектов, некритичных к срокам и качеству выполнения. При этом граница применения компьютеров расширилась до областей, казавшихся ранее недоступными. Хотя конструкторы первых ЭВМ скептически относились даже к будущей возможности компьютерной обработки символьной информации.

Джордж Лукас приступил к созданию недостающих серий «Звёздных войн» только через 20 лет. Ещё дольше ждали создатели первых луноходов и автоматических межпланетных зондов, прежде чем выпустить на разведку роботы-марсоходы. Станки с ЧПУ¹, безлюдные заводы и роботы-хирурги не имели практической возможности воплотиться до 1980-х годов, когда появились массовые достаточно мощные промышленные микропроцессоры. Для многих задач и существующая мощность пока недостаточна, поэтому суперкомпьютеры продолжают её наращивать. Но прогнозы погоды всё равно ошибаются.

Специализация

Исторически сложилось так, что многие программисты были преимущественно математиками и самостоятельно занимались формализацией задач. То есть приведением их к виду, пригодному для решения на компьютере. Сам себе постановщик и кодировщик. В общем виде формула, отражавшая суть работы выражалась так:

Программист = алгоритмизация и кодирование

С ростом сложности задач и упрощения процесса непосредственного кодирования при одновременном усложнении повторного использования кода появилась возможность разделить труд, и формула приобрела примерно такой вид:

¹ Числовое Программное Управление.

Программист минус алгоритмизация = кодировщик
Программист минус кодирование = постановщик задачи

Это вовсе не значит, что мир разделился на аналитиков-алгоритмистов и техников-кодировщиков. Практика многих десятилетий показала, что по-прежнему наиболее востребованным специалистом является инженер, способный как самостоятельно формализовать задачу, так и воспользоваться стандартными средствами её решения на ЭВМ.

Вот его и следует называть программистом.

Современный рынок труда кишит кальками с англоязычных терминов и их комбинациями. Прежде всего, это касается обилия различных видов архитекторов, «девелоперов», «кодёров», «тестеров» и прочих странных прозвищ.

Тестером (не путайте с тостером) раньше назывался прибор-мультиметр, способный измерять напряжение, силу тока и сопротивление. Вы хотели бы работать «тестером»? А если назвать должность в соответствии с её сутью: «инженер-испытатель»? Думаю, отношение к делу сразу изменится.

Чтобы не запутаться в терминологических дебрях жужжащих словечек, сделаем короткое отступление для сопоставления с ранее существовавшей и достаточно понятной всем классификацией.

Кто такой ведущий инженер, или Как это было

Давайте рассмотрим и сравним проектные организации по разработке, поставке и эксплуатации программного обеспечения, оборудования и системной интеграции, условно названные как:

- «А» — организации времён позднего СССР (1960–80 гг.): НИИ¹, КБ², КТЦ³, ВЦ⁴...
- «Б» — современные компании.

¹ Научно-Исследовательский Институт.

² Конструкторское Бюро.

³ Конструкторско-Технологический Центр.

⁴ Вычислительный Центр.

Проектные организации в обоих случаях имеют матричную структуру, то есть работники входят в проект безотносительно административного деления, а именно из разных секторов и лабораторий.

Иерархия подразделений

Уровень	А	Б
1	Организация	Компания
2	Отделение (подразделение)	Дивизион (отделение)
3	Отдел (*)	Отдел
4	Лаборатория (**)	Группа
5	Сектор	Группа

(*) — отдел является единицей финансирования, то есть бюджеты составляются, начиная с уровня отдела;

(**) — уровень лаборатории не являлся обязательным для организаций, не ведущих НИР (научно-исследовательскую работу), например для ВЦКП (Вычислительный Центр Коллективного Пользования).

Иерархия должностей

Уровень иерархии	Образовательный ценз (*)	А	Б	Англоязычный термин
1	Доктор (кандидат) наук	Директор	Директор (генеральный директор)	CEO — Chief Executive Officer
1	Доктор (кандидат) наук	Главный инженер	Технический директор	CTO — Chief Technology/Technical Officer
2	Доктор (кандидат) наук	Заместитель по отделению	Заместитель директора по направлению, директор по направлению	(**)
3	Кандидат наук	Начальник отдела	Начальник отдела	Head of service, head of department

продолжение ⇨

Уровень иерархии	Образовательный ценз (*)	А	Б	Англоязычный термин
4	Кандидат наук	Заведующий лабораторией	Руководитель группы	
5	Высшее образование	Заведующий сектором	Руководитель группы	Team leader
6	Высшее образование	Инженер (категории 3, 2 и 1, старший/ведущий), научный сотрудник (младший, старший)	Специалист, инженер	
7	Среднее специальное образование	Техник, лаборант, студент-практикант	Сотрудник	

(*) — в современных организациях образовательный ценз, как правило, формально не учитывается;

(**) — как правило, укладывается в шаблон Chief «направление» Officer. Например, CAO — Chief Accounting Officer — главный бухгалтер, CDO — Chief Development Officer — директор по развитию и т. д.

Уровни принятия проектных решений

Уровни инвариантны организации, они существуют всегда, но могут быть разными, например, если проект небольшой. Используется иерархия деления: система — подсистема — модуль.

Уровень	Тип принимаемых решений
1	Целеполагание, технико-экономическое обоснование, регламент, бюджет, планирование
2	Требования к системе, концепция, техническое задание, архитектура системы

Уровень	Тип принимаемых решений
3	Технический проект подсистемы, архитектура подсистемы
4	Реализация подсистемы, спецификация модулей
5	Реализация модулей, ввод информации

Функциональные специализации (роли)

Уровень проектного решения	А	Б	Англоязычный термин
1	Руководитель проекта, научный руководитель	Руководитель проекта	Project Manager
2	ГИП (главный инженер проекта), главный конструктор (*)	Системный архитектор	System Architect
3	Ведущий инженер (область, специализация)	Архитектор подсистемы или направления (**)	Software architect, database architect, hardware architect
4	Инженер (область, специализация)	Разработчик (специализация)	Software engineer, database engineer, system engineer
5	Техник (область, специализация), оператор ЭВМ	Младший специалист, кодировщик	Software developer, database developer, web developer, support & helpdesk

(*) — как правило, главный конструктор или ГИП занимали должности от начальника сектора и выше в зависимости от проекта, который они возглавляли;

(**) — уровень архитектора подсистемы/направления соответствует уровню ведущего инженера. Направления специализации могут быть разнообразными: базы данных, человеко-машинный интерфейс, качество (испытания), информационная безопасность, сетевое оборудование, инфраструктура и т. д.

Метаморфозы

Упомянутое в предыдущей главе разделение на инженеров и техников существовало с незапамятных времён. Застал я его «живьём» в конце 1980-х — начале 1990-х годов. В штате институтского ВЦ или конструкторско-технологического центра всегда присутствовали «инженер-программист» и «техник-программист». Инженеры имели категории вплоть до «ведущего», что при совмещении руководства группой соответствовало нынешнему словечку «тим лид» (*team lead*). Техники тоже делились по категориям.

Формальное различие состояло в том, что техников готовили в профильных профессионально-технических училищах (ПТУ) и техникумах; их образование называлось средним специальным. Инженеров готовили технические вузы¹. Наконец, были математики-программисты, которых готовили в университетах.

Фактическое же различие состояло в том, что техники не занимались постановкой задач и проектированием программных систем, ограничиваясь непосредственно программированием и эксплуатацией.

Стандартная должность для программиста-техника называлась «оператор ЭВМ». Некоторое время она сохранялась и после перехода на персональные компьютеры как «оператор ПЭВМ». Потом слово трансформировалась в «эникейщик» (от английского *press any key*) и, позднее, «хелпдеск» (*helpdesk*) — специалист службы поддержки пользователей. Соответственно, системный программист-техник большой ЭВМ превратился в системного администратора по эксплуатации сети «персоналок» и серверов.

Современная ситуация изменилась, нередко диплом о высшем образовании, ранее гарантировавший уровень, достаточный для допуска инженера к проектированию, на практике подтверждает лишь уровень техника. Немало средне-специальных учебных заведений за годы вседозволенности, по недоразумению называемой «либерализацией», стало разными университетами и академиями, и наоборот, некоторые инженерные вузы, чей преподавательский состав отделился от реальных производств и бизнеса, начали выпускать техников с инженерным дипломом. Общее же количество вузов в России с середины 1990-х годов росло как на дрожжах.

Случается наблюдать, как новоиспечённый системой высшего образования программист утверждает: «Мне не понадобилось ничего из того, чем пичкали в институте». Но если эта фраза, произнесенная с гордостью, означает, что

¹ Высшие учебные заведения.

работа никчёмная, то эта же фраза, произнесённая с грустью, говорит о том, что вуз — бесполезный. Поэтому следите за интонациями своей речи, делая подобные заявления.

В Европе, и в частности во Франции, для специалистов с высшим образованием существует чёткое разделение на выпускников инженерных школ и университетов. Первые готовят инженеров для производств, вторые — исследователей для научной и опытно-конструкторской работы. Также негласно считается, что в инженерных школах занимаются серьёзной и целенаправленной подготовкой кадров, тогда как в университетах с их большей внутренней свободой «покуривают травку» между лекциями. Чтобы не объяснять всякий раз новые российские особенности, в результате которых моя альма-матер¹ превратилась из инженерного вуза сначала в академию, а чуть позже и в университет, приходилось в резюме писать прямым текстом «инженерная школа аэрокосмического приборостроения» с устными оговорками о переименовании.

О красоте

Споры на тему, является ли программирование или всё софтостроение в целом искусством, ремеслом или чем-нибудь другим, имеют давнюю историю. Как только разработка программ спустилась с академических вершин на грешную землю, появился широкий слой профессионалов, рассматривающих софтостроение, с одной стороны, как средство заработка на жизнь, а с другой — как средство реализации своих идей в техническом творчестве.

Хороший термин — «техническое творчество». От него веет полузабытой атмосферой школьных кружков, где была написана первая программа или смонтировано первое роботоподобное устройство. Те ученики давно подросли, стали профессионалами, но умудрились сохранить творческий подход к делу. А поэтому, как бы ни стандартизировали отрасль, эти люди постараются найти место для реализации своих идей. Сделают не просто «чтобы работало», а чтобы ещё и «было красиво».

Учёный и классик жанра научной фантастики Иван Ефремов писал: «Красота — это высшая степень целесообразности в природе, степень гармонического

¹ Неформальное название учебных заведений.

соответствия и сочетания противоречивых элементов во всяком устройстве, во всякой вещи и во всяком организме».

Нельзя «сделать красиво», если относиться к работе исключительно утилитарно и шаблонно. Получаются сплошные типовые дома и «мыльные» сериалы. Необходимы и чувство прекрасного, и чувство меры, и знание других образцов, считающихся лучшими. Нужна техническая культура. Долгая работа, неблизкий путь, мотивация преодолеть который исходит, прежде всего, от любви к собственному делу, к профессии.

Но и нельзя «сделать красиво», если рассматривать софтостроение лишь как искусство и средство самовыражения. Любить себя в софтостроении, а не софтостроение в себе. Тогда красота рискует так и остаться не воплощёнными в жизнь эскизами. Невозможно обойтись без знаний технологий производства и хороших ремесленных навыков.

Пока одни корпели над программами и моделями в кружках, другие реализовывали свои интересы, вполне возможно, творческие, но не технические. И вот в связи с процессом заполнения сферы услуг, о котором мы ещё поговорим, эти другие тоже оказались в софтостроении, поскольку имеется устойчивый спрос на рабочую силу. Разумеется, для таких работников главной, а то и единственной целью будет сделать «чтобы как-то работало». Это даже не ремесло в чистом виде, а неизбежные плоды попыток индустриализации в виде халтуры и брака.

В итоге программирование нельзя целиком причислить ни к искусству, ни к ремеслу, ни к науке. Софтостроение на текущий момент — эклектичный сплав технологий, которые могут быть использованы как профессионалами технического творчества, так и профессионалами массового производства по шаблонам и прецедентам. Поскольку наука все больше отдаляется от софтостроения, то предсказать, что выйдет в каждом конкретном случае — архитектурный шедевр, типовой панельный дом или коровник, практически невозможно. Кадры решат всё.

6 миллионов на раздел пирога

В повести Аркадия и Бориса Стругацких «Гадкие лебеди» есть примечательный фрагмент диалога между писателем Виктором Баневым и школьниками, пригласившими его на встречу:

— Разрешите мне, — сказал Бол-Кунац. — Давайте рассмотрим схему. Автоматизация развивается в тех же темпах, что и сейчас. Только через несколько десятков лет подавляющее большинство активного населения земли выбрасывается из производственных процессов и из сферы обслуживания за ненадобностью. Будет очень хорошо: все сыты, топтать друг друга не к чему, никто друг другу не мешает... И никто никому не нужен. Есть, конечно, несколько сотен тысяч человек, обеспечивающих бесперебойную работу старых машин и создание новых, но остальные миллиарды друг другу просто не нужны. Это хорошо?

— Не знаю, — сказал Виктор. — Вообще-то это не совсем хорошо. Это как-то обидно... Но должен вам сказать, что это все-таки лучше, чем то, что мы видим сейчас. Так что определённый прогресс все-таки налицо.

Со времён написания повести прошло 40 лет, нарисованная школьником схема стала реальностью. Производительность труда растёт, высвобождающиеся из производственных цепочек люди вынуждены уходить в сферу услуг. Но и она не бездонна. Придумывать и выводить на рынки всё более изощрённые занятия вроде моделирования костюмов для домашних животных становится все труднее.

В Германии совершенно серьёзно и открыто обсуждается идея выплаты всем гражданам безусловного основного дохода¹ (БОД) — минимального пособия примерно в 1000 евро, которого хватит на оплату недорого жилья, скромного питания и одежды. Пособие бессрочное и выплачивается всем гражданам независимо от того, есть у них работа или нет. Те же, кто работает, должны получать заработную плату в качестве добавки к БОД. Во Франции похожее пособие (*Revenu de Solidarité Active*) существует достаточно давно, с 1988 года, но касается только уже потерявших право на выплаты по безработице; при этом размер пособия порядка 400 евро недостаточен для аренды жилья.

Как определить, любите ли вы свою работу, профессию, дело, которым заняты? Представим на минутку, что вы живёте в Германии и получаете свой БОД. То есть каждому дают по минимальным потребностям, а по способностям — не спрашивают. Ответьте себе честно. Имея возможность потреблять, не отдавая взамен свой труд, останетесь ли вы профессионалом в своей сфере?

¹ Оригинальное немецкое название: *Bedingungsloses Grundeinkommen*.

Вопрос неспроста. Недалеко от нашего городка есть так называемый «ассоциативный гараж». То есть мастерская общего пользования местных авто- и мотолюбителей, где они самостоятельно могут выполнить осмотр и небольшой ремонт своей машины. По субботам в гараж приходят бывшие профессионалы, ныне находящиеся на предпенсионной программе или недавно вышедшие на пенсию. Не секрет, что при европейском качестве жизни к 60 годам многие мужчины всё ещё полны сил и желания работать в своё удовольствие. Они охотно и совершенно бесплатно помогают автолюбителям советами и делом. Для автомастерских в округе подобная ситуация приносит одни убытки. В результате мэрия вынуждена ограничить работу гаража одним днём в субботу до полудня. Если бы такая нелояльная конкуренция продолжалась, то гараж попросту сожгли бы.

Подобная конкуренция среди программистов имеет некоторые отличия, о которых мы и поговорим.

Круговорот

Софтостроение представляет собой соединение относительно небольшого сегмента продуктового производства массово тиражируемых системных сред, средств разработки, прикладных систем, пакетов и огромного рынка услуг, связанных с этими продуктами. Я бы оценил их соотношение как 10 % к 90 %, но, боюсь, такая пропорция будет слишком оптимистичной.

На практике это означает, что на одного производителя тиражируемого продукта разной степени серийности — от массовых брендов до малотиражных специализированных, приходится почти десяток поставщиков услуг, крутящихся вокруг этих продуктов, и разработчиков заказных программных систем. Чем дальше от основного производителя программных продуктов в мире — США, тем больше это соотношение в пользу сервиса.

Уникальность софтостроения как сферы услуг в его высокой кадровой ёмкости. Представьте себе отдел «Х» в управлении крупной фирмы, использовавший для решения какой-то задачи электронные таблицы офисного пакета. На основном производстве тем временем внедрили новую технологию, снизив издержки и сократив несколько работников.

Куда направить освободившиеся ресурсы? А давайте-ка автоматизируем непроизводительную возню отдела «Х» с таблицами, и тогда начальники смогут быстрее получать сводки!

Запускается проект. Он не критичен по срокам и качеству. Критичные системы уже давно в эксплуатации, и трогать их никто в здравом уме не будет. Своих программистов в компании нет — непрофильная деятельность. Поэтому заказ спокойно направляют в проектно-консультационную фирму, где, кстати, вполне могут работать выведенные лет 10 назад за штат собственные программисты. У подрядчика, занятого поддержкой существующих систем, может не хватить ресурсов на новый проект, и он вывесит вакансию.

Тем временем уволенные с основного производства приходят в службу занятости, где им говорят: «Специалистов вашего профиля повсюду сокращают. Предлагаем вам переквалификацию». И дружными рядами бывшие операторы устаревшей автоматизированной линии идут на трёхмесячные курсы «Разработка приложений в среде Basic» или «Разработка веб-приложений». После окончания учёбы они попадают на работу в фирму-подрядчик, где начинают автоматизировать использование электронных таблиц отделом компании, откуда их несколько месяцев назад сократили.

Вот такой, если очень упрощенно, происходит круговорот.

Возникает естественный вопрос: «А как же конкуренция по себестоимости разработки, которая должна двигать прогресс в отрасли?»

Конкуренция, конечно, формально существует. Но если в производстве она тесно связана со снижением издержек, то в сфере услуг на первый план выходят доверительные отношения между заказчиком и подрядчиком, снижающие риски. Кроме того, проекты нетиповые, заказные, и найти еще одного подрядчика, уже имеющего аналогичный опыт, — это новые затраты и риски. У существующего подрядчика, сопровождающего программы, может быть договор на приоритет новых заказов. Ведь новые программы должны интегрироваться с уже работающими — это опять вопрос отношений с проверенным поставщиком, а попытка сталкивать его лбом с конкурентом может выйти боком вообще всем участникам процесса. Масса скрытых особенностей, непрозрачная среда, полная корпоративных игр и политики. У европейцев есть на сей счёт соответствующая оговорка, что непосредственная власть находится в руках управленцев среднего звена.

На практике замена старого подрядчика новым — весьма рискованная процедура даже на некритичных проектах. Потребуются немалые затраты при неочевидности выгод обмена «шила на мыло». Тогда как менеджеры стремятся, с одной стороны, затраты, наоборот, сократить, а с другой — максимально раздуть бюджет и штат для его освоения ради продвижения по карьерной лестнице. Вот такие противоречивые задачи постоянно вынужден решать менеджер.

Круговорот вовлечения в сферу услуг исключённых из производственных цепочек людей касается не только программистов. За последние два десятилетия практически все крупные западные компании «экстернализовали», то есть вывели за штат, большинство специалистов из отделов информационных технологий. Инфраструктура приложений — серверы, программное обеспечение, администрирование, безопасность — всё поддерживается подрядчиками. В штате остаётся минимум ответственных за связь с подрядчиками и обслуживание парка компьютеров.

Разница в том, что инфраструктурные услуги неплохо оптимизируются и один бывший администратор сети или баз данных компании может теперь обслуживать сразу несколько клиентов, работая удалённо на прямой связи, а то и непосредственно в ВЦКП¹ или ЦОД².

В софтостроении такая оптимизация оказывается проблематичной. Кроме упомянутых проблем с конкуренцией, имеет место и другая веская причина — нечёткость требований, сформулировать которые заказчик далеко не всегда в состоянии. Ведь, как вы помните, он уволил своих прикладных программистов ещё 10–15 лет назад. У подрядчика же функциональная специализация программистов существует только для клиентов, способных давать стабильный заказ с высокой долей прибыли. В первую очередь, это банки и финансовые компании. Проектная фирма обычно вкладывает собственные средства в обучение разработчиков предметной области таких заказчиков, вплоть до получения ими второго высшего образования. Найти же программистов, знающих специфику работы отдела «Х» с электронными таблицами, мягко говоря, маловероятно. Подойдёт и бригада после курсов переквалификации, возглавляемая более опытным руководителем, скорее всего, имеющим не техническое, а коммерческо-управленческое образование.

Мельница крутится, в разработку «проектов для отделов «Х» и следующую за этим через несколько лет переделку втягивается всё больше людей. Можно с уверенностью сказать, что писавших программы в школьных кружках среди них нет, поскольку такой специалист изначально работал бы в софтостроительной сфере. Хорошо, если они вообще имеют техническое образование. На курсах же дают только некоторый набор приёмов, за счёт которого, постепенно расширяя арсенал, им придётся зарабатывать себе на жизнь. Если

¹ Вычислительный Центр Коллективного Пользования.

² Центр Обработки Данных (англ. Data Center).

голова работает нормально, то бывший новичок за несколько лет превращается в крепкого ремесленника с перспективой сопровождения своих программ до заслуженной пенсии.

Масштабы и последствия

Согласно сведениям IBM, сообщество Java-разработчиков уже к 2006 году насчитывало более 6 миллионов человек¹. Вдумайтесь в эту цифру. Шесть миллионов ремесленников ежедневно садятся перед монитором и усердно вбивают в дисковое пространство программный код.

Когда вступают в действие большие числа, впору вспомнить о нормальном распределении, на которое нам открыл глаза ещё старина Гаусс.

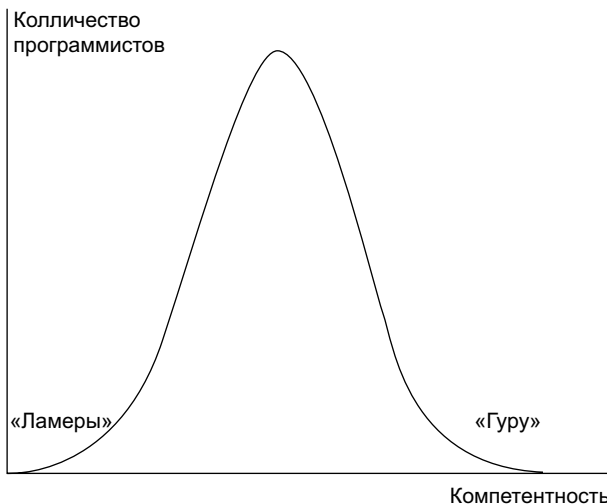


Рис. 1. Нормальное распределение уровня профессиональной компетентности программистов

Чтобы не просто зарабатывать на хлеб, но и мазать его маслом, сохраняя при этом возможности технического творчества, вам лучше держаться подальше от тех направлений деятельности, где конкурентами будут 6 миллионов человек.

¹ «With today's news, the companies are reinforcing their commitment to the Java community, which comprises more than six million developers worldwide» // IBM Taps Boom in Linux Growth by Expanding Commitment to Partners, Linux and Open Source, december 2005.

И отнюдь не из-за охлофобии. Огромное количество программистов, в первую очередь, означает, что данная технология вполне доступна не только для «среднячков», на которых мир держится, но и для откровенных дилетантов. Я даже уверен, что среди дилетантов процент честно заучивающих имена местных «гуру», жаргон и прочие «паттерны» выше, чем среди остальных — для них это, прежде всего, вопрос прохождения интервью.

Большое количество дилетантов нивелирует строчки в резюме и профессиональные сертификаты в глазах заказчика или работодателя, несмотря на опыт и представленные проекты.

Я не верблюд, чтобы доказывать, что я не верблюд.

Когда выборка составляет 6 миллионов, несложно получить и среднюю по отрасли оплату своего труда. И можно себе представить, скольких усилий стоит добиться высокой оплаты. Не будет иметь большого значения то, что ты можешь сделать хороший дизайн, если за тобой на интервью придёт дилетант, заучивший десять известных работодателю «паттернов», 200 классов фреймворка¹ и просящий за это в 2 раза меньше денег.

Отсюда неутешительный вывод для писавших программы в школьных кружках: количество проектов, где потребуется ваша квалификация, намного меньше количества некритичных заказов, а большинство ваших попыток проявить свои знания и умения столкнется с нелояльной конкуренцией со стороны вчерашних выпускников курсов профессиональной переориентации. На практике это означает, что вам, возможно, придётся снижать цену своего труда и готовиться к менее квалифицированной работе.

Не забывайте, что относительная доля критичных к качеству проектов падает, а переделка работающих систем базового уровня, от которых непосредственно зависит бизнес, — и вовсе редкое явление. Этот момент всегда оттягивают до последнего, предпочитая использовать вышедших на пенсию кобол-программистов и модернизацию мейнфреймов² с помощью специалистов IBM. Слишком

¹ От англ. framework. В рамках объектно-ориентированного подхода — библиотека классов с двусторонним (взаимным) управлением потоком исполнения программы. Более общее значение — каркас, предоставляющий стандартные службы, библиотеки и компоненты для разработки программ в рамках накладываемых им ограничений.

² От англ. mainframe — классическая большая универсальная ЭВМ.

высоки риски. Новые значимые проекты возникают только с новыми рынками и направлениями бизнеса.

Поэтому немало специалистов высокой квалификации уходят в экспертизу и консалтинг, где проводят аудит, обучение, «натаскивание» и эпизодически «вправляют мозги» разным группам разработчиков из числа переквалифицировавшихся.

Да, можно найти проект с уже набившими шишек заказчиками и квалифицированными менеджерами, но места для поиска можно пересчитать по пальцам. И тогда это в принципе мало отличается от работы с узкой специализацией на технологиях. По-прежнему, разработка программ параллельных вычислений, разработка алгоритмов защиты и шифрования или системное администрирование UNIX требуют кадры, которые курсы переквалификации выдать не могут.

Другой доступный вариант — специализация на предметных областях. В этом случае разработчик относительно автономен и, во-первых, гораздо менее ограничен в выборе инструментов. Во-вторых, что более существенно, доказывать кому-то степень владения инструментарием у него нет необходимости. К сожалению, хорошее знание предметных областей в сочетании с глубокими техническими знаниями платформ встречается редко в связи с плохой совместимостью высокоуровневых абстракций и низкоуровневых деталей. Обычная эволюция такого специалиста — системный аналитик, сохранивший знания технологий времён своего последнего сеанса кодирования в интегрированной среде.

Хорошо оплачиваемая работа с творческим подходом к труду в современном мире — это привилегия, за которую придётся бороться всю жизнь. И софтостроение здесь не является исключением.

Профориентация

В средней школе многие проходили профориентационные тесты по классификации Климова. Помните, «человек—человек», «человек—техника»? Сколько из вас тогда попало в категорию «человек—человек»? В нашем классе совершенно обычной ленинградской средней школы таковых было менее трети. Немного позднее в классе математической школы тот же самый тест дал ещё меньший результат.

Как вы помните, софтостроение на 90 % находится в сфере услуг. Если вы не работаете на производстве у одного из поставщиков тиражируемого программного обеспечения, то взаимодействия типа «человек—человек» становятся необходимым и важным элементом повседневной работы, если только вы не предполагаете всю жизнь провести в кодировании чужих спецификаций, не всегда толковых и формализованных. Вместо решения сугубо технических задач вроде оптимизации конфигурации версий продукта для разных типов клиентов, вашей целью будет решение задач конкретных клиентов. А критерием решения станет субъективная степень удовлетворённости клиента.

Во французском языке существует специальный термин «чувство службы» (*sens de service*). В русском языке также имеется старинное полузабытое слово «*услужливый*». Чтобы работать софтостроителем в сфере услуг, нужно, простите за тавтологию, уметь быть услужливым. В ещё большей степени «чувство службы» касается консультантов.

Например, когда для публикации функции подключаемого модуля (*plug-in*) в меню основного приложения требуется тем или иным образом её декларировать в семи-восьми разных местах, эксперт-аудитор с «чувством службы» вместо нецензурной лексики пишет «несомненно, эта ситуация стала следствием сложных обстоятельств развития системы и не является прямой ошибкой проектировщиков».

Малозначимым в глазах руководства может оказаться не только проект, но и обслуживание крупного продукта. Весьма показательный уровень программирования в одной социальной сети можно было оценить по пришедшему от их имени письму следующего содержания: *«Ваши фотографии были перенесены на наш новый фотохостинг. Всего было перенесено 0 фотографий»*. Для того чтобы вставить в код программы рассылки проверку $IF > 0$, нужно, видимо, иметь не только недюжинные умственные способности, но и дополнительную квалификацию, равно как и понимание сути выполняемой задачи. С другой стороны, да и чёрт с ними, с сотнями тысяч отправленных бесполезных писем. Одной массовой рассылкой больше, одной меньше, не правда ли?

В постиндустриальной экономике сфера услуг занимает более 50 % деятельности, и эта доля растёт, например, в США она уже близка к 70 %. Представьте себе ваш бывший школьный класс, где к работе в обслуживании ориентированы не более 25 %. Откуда же брать недостающих, да при этом еще и услужливых? Проблема, удовлетворительных решений которой на сегодняшний день не найдено. Поэтому и обсуждают введение пособий типа БОД: пусть лучше получают

свой минимум и занимаются чем хотят, чем портят отношения с клиентами, мешая производительному труду остальных.

Начинающим соискателям

Для начинающих я составил небольшой словарь ключевых фраз, часто присутствующих в объявлении о вакансии. По замыслу, он должен помочь молодому соискателю вакансии программиста разобраться в ситуации и принять решение на основе более полной информации:

1. «Быстро растущая компания» — фирма наконец получила заказ на нормальные деньги. Надо срочно нанять народ, чтобы попытаться вовремя сдать работу.
2. «Гибкие (*agile*) методики» — в конторе никто не разбирается в предметной области на системном уровне. Программистам придётся «гибко», с разворотами на 180 градусов, менять свой код по мере постепенного и страшного осознания того, какую, собственно, прикладную задачу они решают.
3. «Умение работать в команде» — в бригаде никто ни за что не отвечает, документация потеряна или отсутствует с самого начала. Чтобы понять, как выполнить свою задачу, требуются объяснения коллег, как интегрироваться с уже написанным ими кодом или поправить исходник, чтобы наконец прошла компиляция модуля, от которого зависит ваш код.
4. «Умение разбираться в чужом коде» — никто толком не знает, как это работает, поскольку написавший этот код сбежал, исчез или просто умер. «Умение работать в команде» не помогает, проектирование отсутствует, стандарты на кодирование, если они вообще есть, практически не выполняются. Документация датирована прошлым веком. Переписать код нельзя, потому что при наличии многих зависимостей в отсутствии системы функциональных тестов этот шаг мгновенно дестабилизирует систему.
5. «Гибкий график работы» — программировать придётся «отсюда и до обеда». А потом после обеда и до устранения всех блокирующих ошибок.

6. «Опыт работы с заказчиком» — заказчик точно не знает, чего хочет, а зачастую — неадекватен в общении. Но очень хочет заплатить по минимуму и по максимуму переложить риски на подрядчика.
7. «Отличное знание XYZ» — на собеседовании вам могут предложить тест по XYZ, где в куске спагетти-кода нужно найти ошибку или объяснить, что он делает. Это необходимо для проверки пункта 4. К собственному знанию XYZ-тест имеет очень далёкое отношение.

Тесты — особый пункт при найме. Чаще всего они касаются кодирования, то есть знания синтаксиса, семантики и «что делает эта функция».

Много лет назад по необходимости я составил небольшой сборник подобных тестов по Delphi и Transact SQL для соискателей вакансии программиста. Время от времени пользовался. Частенько люди, не сумевшие ответить на большинство вопросов, просили тесты забрать с собой. Забирайте, на здоровье.

Спустя десяток лет посмотрел на те же тесты скептически. Знание технологии они ещё худо-бедно позволяют выяснить, а вот как человек мыслит — непонятно. Мой опыт говорит, что «натаскать» можно практически на любой формальный тест даже «двоечника». Поэтому не стоит мерить интеллект тестом на IQ¹. Лучше давать испытуемому некоторый нестандартный тест, чтобы просто посмотреть на ход его мысли. Или давать один такой тест 2 раза подряд, выводя IQ не из результатов, а из прогресса верных ответов на второй итерации. Неисчерпаемым источником для неформальных тестов может послужить полная нестандартных задач книга профессионального системного программиста Чарльза Уэзерелла «Этюды для программистов» [17].

Про CV

Cirriculum Vitae, или CV, оно же по-русски «резюме», является важной деталью вашего представления потенциальному работодателю. На Западе принято прикладывать к нему ещё и мотивационное письмо, об искусстве написания которого выпускаются целые брошюры. Поэтому ограничимся только CV.

Я сформулировал бы основные принципы хорошего резюме следующим образом:

¹ Коэффициент интеллекта (англ. intelligence quotient).

- **Краткость — сестра таланта.** Даже в небольшой фирме ваше резюме будут просматривать несколько человек. Вполне возможно, что первым фильтром будет ассистент по кадрам, который не имеет технического образования и вообще с трудом окончил среднюю школу. Поэтому постарайтесь на первой странице поместить *всю* основную информацию: ФИО, координаты, возраст, семейное положение, мобильность, личный сайт или блог, описания своего профиля, цель соискания, основные технологии с оценкой степени владения (от «применял» до «эксперт»), образование, в том числе дополнительное, владение иностранными языками. Всё остальное поместите на 2–3 страницах.
- **Кто ясно мыслит, тот ясно излагает.** Все формулировки должны быть ёмкими и краткими. Не пишите «узнавал у заказчика особенности некоторых бизнес-процессов в компании» или «разработал утилиту конвертации базы данных из старого в новый формат». Пишите «занимался постановкой задачи» или «обеспечил перенос данных в новую систему».
- **Не фантазируйте.** Проверьте резюме на смысловые нестыковки. Если на первом листе значится «эксперт по C++», но при этом в опыте работы за последние 5 лет эта аббревиатура встречается один раз в трёх описаниях проектов, то необходимо скорректировать информацию.
- **Тем более не врите.** Вряд ли кадровики будут звонить вашим предыдущим работодателям, но софтостроительный мир тесен, а чем выше квалификация и оплата труда, тем он теснее. Одного прокола будет достаточно для попадания в «чёрный список» компании, а затем через общение кадровиков и агентств по найму — ещё дальше.
- Если соискание касается технического профиля, в каждом описании опыта работы **упирайте на технологии**, если управленческого — на периметр ответственности, если аналитического — на разнообразие опыта и широту кругозора.
- **Не делайте ошибок.** Пользуйтесь хотя бы автоматической проверкой грамматики. Мало того, что ошибки производят негативное впечатление, они могут радикально изменить смысл фразы. Например, если написать «политтехнический университет»...

- **Будьте готовы, что далее первой страницы ваше резюме читать не станут**, а о подробностях «творческого пути» попросят рассказать на первом собеседовании.

После обсуждения вашего сносшибательного CV и ответного рассказа работодателя о том, как «космические корабли бороздят просторы их малого или большого театра», соискателю, как правило, следует что-нибудь спросить. Вот мой вариант. Я постарался быть краток:

Соискатель: Вы используете так называемые «гибкие» методы, например Scrum? Если да, то какова степень формализации процесса? У вас есть аналитики и проектировщики? Какие модели вы используете? Есть ли практика ежедневных утренних планёрок? Есть ли ответственные за подсистемы?

Работодатель: Да — высокая — выделенных нет — что-то рисуется в UML — обязательно! — есть, трудовой коллектив.

Соискатель: Спасибо, всего вам доброго и успехов в труде!

Про мотивацию

Мне очень нравится одна история-притча, которую приведу целиком:

Около дома одного человека мальчишки играли в мяч: ударяли им о стены, громко кричали и смеялись. Естественно, они мешали хозяину дома. И вот в один прекрасный день он вышел к ним и сказал: «Друзья, вы так весело играете в мяч, так заразительно смеётесь и кричите, что я с удовольствием вспоминаю свое детство. Я буду платить каждому по монете, чтобы вы каждый день приходили сюда, громко кричали, смеялись и играли в мяч». Мальчишки взяли по монете и продолжили игру. На следующий день они снова пришли и получили по монете. Так продолжалось несколько дней. Но как-то хозяин подошёл к мальчишкам и сказал, что его финансовые дела не так хороши, как раньше, и он сможет платить им только по полмонеты. Он заплатил им по полмонеты и ушёл. А мальчишки поговорили и решили, что не будут стараться за полмонеты. И больше они не приходили. Так хозяин дома получил желаемые мир и спокойствие...

Трудно сказать, почему вместо слова «стимуляция» повсеместно прижилось «мотивация». Навязли в зубах рекламные рассылки «способы мотивации персонала», которая совсем не мотивация, а стимуляция. Не иначе, консультанты хотят избежать нежелательных ассоциаций со стимулированными свинками и собаками Павлова. Или с древнеримской палкой-стимулом, при помощи которой помыкали домашним скотом.

Мотивация — исключительно внутренний механизм. Чтобы управлять им, необходимо залезать в этот самый механизм, в психику. Существуют традиционные способы: педагогика и воспитание. Они требуют многих лет, а то и смены поколений. Существуют и более быстрые варианты, связанные с химией и традиционной медициной. Наконец, есть и очень быстрые и рискованные, связанные с психотехниками, гипнозом и «зомбированием».

Поэтому, когда вам предлагают услуги по «мотивации персонала», желательно спросить, будут ли персонал бить током, колоть препаратами или ограничатся лёгким массовым сеансом гипноза.

Напротив, *стимуляция* — это внешний механизм. Он основан на выявлении мотивов и последующем их поощрении или подавлении. Стимулятор подобен катализатору для запуска химической реакции. Управление стимуляцией сводится к созданию системы стимулов, требуемых для «реакций» катализаторов. Для «реакций» — процессов работы человеческих коллективов.

Управлять мотивацией, то есть целенаправленно изменять психологию и выстраивать набор стимулов — это «две большие разницы». Первое, по сути, требует изменения самих людей, второе — это использование имеющихся у них мотивов. Мотивировать же можно только свои поступки, но никак не образ действия окружающих.

Изгибы судьбы при поиске работы

Рассказ из реальной жизни, надеюсь, внесёт долю юмора в оказавшуюся не самой весёлой тему профессиональной ориентации.

С наступившей весной я озаботился сменой работодателя. Действительно, на дворе кризис, «троечники» сидят по своим местам, самое время поискать весёлую компанию «отличников» или, на худой конец, «хорошистов». Правда, время поиска вырастает раза в 2, но в течение месяцев так трёх-четырёх найти место вполне реально

даже при финансовых запросах выше среднего. Технология ловли рыбы в мутной воде несложная, достаточно разместить резюме на одном из крупных веб-сайтов, параллельно входя напрямую в контакт с отдельными компаниями.

Как обычно, регулярно названивали мадемуазельки ранга ассистента по «человечьим ресурсам», первый их вопрос стандартен, вызван дилетантизмом в предметной области нанимаемых и потому звучит всегда: «А какую работу вы ищете, если поточнее?» То есть объясните, дяденька, что это за аббревиатуры такие у вас на первой странице CV. Если настроение хорошее, можно коротко просветить девушку, если не очень, то отвечать в стиле: «Ровно то, что написано в заголовке CV, вы его читали?»

Второй этап — выяснение, ищут ли они специалиста на конкретный проект или просто под широкий профиль. Это вопрос специфичный для самого массового работодателя — консультационно-проектных фирм. Если набор идёт «под широкий профиль», то можно вежливо начинать прощаться.

Чтобы избежать этого этапа, а также при желании найти место у конечного клиента, следует ограничить круг своего общения кадровыми агентствами или просто не указывать в резюме номер телефона, ограничившись электронной почтой.

Третий этап — отбрыкаться от приглашения на бесполезное интервью, убедив, что не стоит зря тратить время. Ведь у мадемуазелек рабочего времени навалом. Достаточно попросить прислать краткое описание требований к вакансии. В 90 % случаев это срабатывает, причём в 90 % из этих 90 % случаев требования оказываются неподходящими. В оставшихся 10 % можно соглашаться на интервью, главное — потом накануне не забыть уведомить, что по уважительной причине прийти на него нет никакой возможности.

Ладно, это все техника, которая приходит с опытом. Тем более, если времени много, например, вы сидите на пособии по безработице — это надо же так постараться в нашем расцвете лет, можно и походить по мадемуазелям, расширив круг повседневного общения.

Есть в округе довольно крупная проектная контора, назовём её «Контрабас», несколько тысяч человек, входит во французскую «де-

сятку». Я всячески избегаю крупных фирм общего профиля, потому что уровень технической экспертизы там весьма посредственный при ожидаемом беспорядке в управлении, с многодневным прохождением нескольких уровней бюрократии для простейших операций вроде покупки билета на скоростной поезд, буксующей из-за отсутствия названия станции назначения в корпоративной системе управления.

Как раз звонят из этой конторы. Но не ассистентка, а паренёк, и, видимо, подкованный. Так как после первой минуты сказал, что ещё через четверть часа мне перезвонит собственно начальник отдела, куда ищут работника. С руководителем мы приятно побеседовали минут 20. Выяснилось, что хотя профиль и не совсем тот, что нужен «ещё вчера», но вот очень скоро будет проект и уж там-то... Ну и хорошо, как будет, так и созвонимся-спишемся? Спишемся.

Вечером в почтовый ящик падает анкета «Контрабаса» для соискателя на 10 (!) страницах. Я тут же ее закрыл, письмо переместил в корзину.

Ещё через пару недель позвонила не просто мадемуазелька, а уже целая мадам. Из той же конторы, но из другого подразделения. После фраз о том, как много у них вакансий по моему профилю и приглашения на интервью, пришлось отвечать, что прийти я смогу только для разговора по конкретной вакансии, а не по их множеству. Мадам несколько впала в ступор и в течение пары минут переспрашивала и объясняла, что у них вот такая процедура найма и никак иначе нельзя. Мне было очень жаль, но раз такая процедура найма — тем хуже для найма.

Третий акт интермедии произошёл уже после того, как я нашёл себе небольшую контору человеческого размера из бывших писателей программ в школьных кружках и вышел на работу. Оказалось, что «Контрабас» с малопонятными целями умудрился купить мою новую контору.

В конце концов, по сумме обстоятельств я стал сотрудником «Контрабаса», избежав заполнения 10-страничной анкеты и нескольких раундов интервью, из которых смысл имеет только один — с непосредственным начальником или напарниками, но остальные можно не пройти по совершенно не зависящим от тебя причинам. Например, много лет назад я по неопытности пытался объяснить мадемуазельке

из фирмы-посредника разницу между SQL и PL/SQL, потому что это было важно для данной вакансии. А она только улыбалась. Но по итогам выдала моему агенту заключение: «Не могу рекомендовать вашего инженера клиенту, он был со мной очень холоден...» Я не шучу, формулировка была именно такой.

Коллеги, не будьте холодны с мадемузельками-ассистентками из кадровых служб! Уважайте их заслуженное право на какую-то деятельность после с трудом законченной средней школы и курсов.

Технологии

Никогда не догоняйте
устремившихся вперёд.
Через пять минут, ругаясь,
Побегут они обратно,
И тогда, толпу возглавив,
Вы помчитесь впереди.

Г. Остер

Термин «гуглизация» (*googlization*) не случайно созвучен с другим, с глобализацией. И если глобализация систематично уничтожает закрытые экономики, то «гуглизация» нивелирует энциклопедические знания. Пространство практического применения эрудита сузилось до рамок игры «Что? Где? Когда?». Доступность информации снизила ее значимость, ценность стали представлять не сами сведения из статей энциклопедии, а владение технологиями.

Часть пролетариата умственного труда, способная хранить и воспроизводить технологии, превратилась в «когнитариат». Появился термин «индустриальная археология», касающийся реинжиниринга¹ систем в промышленной эксплуатации, принципы и технологии работы которых неизвестны никому из обслуживающего их персонала.

Технологии в аппаратном обеспечении, «железе», подчинены законам физики, что делает их развитие предсказуемым с достаточной долей достоверности. Зная, какие работы ведутся в лабораториях, можно предугадывать потолок их развития и предполагать сроки готовности к практическому использова-

¹ От англ. «reverse engineering» — восстановление общих проектных решений и концепций по имеющейся частной их реализации.

нию. Например, сейчас в активной фазе находятся прикладные исследования по созданию масштабируемой технологии проектирования и производства устройств, способных заменить нынешние полупроводниковые схемы. Вывод: через десятилетие мир вычислительных устройств изменится.

В противоположность этому, мир программных технологий основан на математических и лингвистических моделях и подчинён законам ведения бизнеса. Крупные капиталовложения, сделанные в существующие средства разработки, инфраструктуру и обучение пользователей, должны окупаться независимо от значения синуса в военное время, релятивистских поправок и элементной базы ЭВМ. Вывод: радикальных изменений в софтверостроительной сфере ожидать не следует, ситуация находится под чутким контролем крупных корпораций и развивается эволюционно.

Тем не менее в софтверостроении, даже кустарном и далёком от индустриализации, технологии составляют основу. О них мы и поговорим.

Можно ли конструировать программы как аппаратуру?

Для развития аппаратной части определяющими являются физические законы, а основой индустриализации в производстве «железа» стали проектирование и сборка устройств из стандартизованных компонентов.

Конечно, в софтверостроении тоже имеются относительно стандартные подсистемы: операционные среды, базы данных, веб-серверы, программируемые терминалы и тому подобное. Однако их масштаб соответствует не компоненту в устройстве, а достаточно сложной аппаратной подсистеме вроде маршрутизатора или сервера.

Возможность собирать изделия из «кубиков» стала предметом зависти софтверостроителей, вылившейся в итоге в компонентный подход к разработке. Панацеи, разумеется, не получилось, несмотря на серьёзный вклад технологии в повторное использование «кубиков», оказавшихся скорее серыми ящиками с малопонятной начинкой. Но появился целый рынок, где писатели компонентов предлагают свои изделия «компонентокидателям» — это жаргонное слово возникло в среде наиболее массового применения компонентов, где их

выбирают на палитре мышкой и, протаскивая, кидают¹ на разрабатываемую экранную форму.

Для аппаратуры используется модель конечного автомата. Во-первых, она обеспечивает полноту тестирования. Во-вторых, компонент работает с заданной тактовой частотой, то есть обеспечивает на выходе сигнал за определённый интервал времени. В-третьих, внешних характеристик (состояний) у микро-схемы примерно *два в степени количества «ножек»*, что на порядки меньше, чем у программных «кубиков». В-четвёртых, высокая степень стандартизации даёт возможность заменить компоненты одного производителя на другие, избежав сколько-нибудь значительных модификаций проекта.

В софтверостроении использовать конечно-автоматную модель для программного компонента можно при двух основных условиях:

- Программисту не забыли объяснить эту теорию ещё в вузе (см. выше про «Круговорот»).
- Количество состояний обозримо: они, как и переходы, достаточно легко определяются и формализуются.

Второй пункт более важен. На практике количество состояний даже несложного модуля запредельно велико, поэтому программист использует их объединения в группы и применяет различные эвристики для обеспечения желаемого результата на выходе при заданном входе.

Возьмём относительно простой пример: компонент, конвертирующий сумму из одной валюты в другую.

Из элементов стандартизации точно присутствуют коды валют по ISO 4217² и, частично, список служб, к которым компонент может обращаться (см., например, каталог служб *Financial API*). Интерфейс самого компонента не стандартизован, для возможной его замены в будущем без последующей структурной перекройки вашего приложения потребуеться обернуть компонент в адаптер (привет, шаблоны!). Это поможет избежать реструктуризации при замене, но не гарантирует работоспособность на том же входном наборе.

¹ Англ. Drag and drop.

² Международный стандарт, регламентирующий кодификацию валют.

Теперь оценим количество состояний, которые необходимо охватить для полноты модульного тестирования, раз уж мы следуем логике разработки «железа». ISO 4217 даёт список из 164 валют. Предположим, что наши входные данные:

- имеют только два знака после запятой;
- значения положительные;
- максимальная величина — 1 миллион;
- дата конвертации всегда текущая;
- мы используем только 10 валют из 164.

Несложный комбинаторный подсчёт показывает, что даже такой сильно урезанный входной набор характеризуется количеством размещений из 10 по 2, помноженным на 100 миллионов входных значений (1 миллион с шагом 0,01):

$$10^2 \times 100\,000\,000 = 10\,000\,000\,000.$$

То есть для обеспечения полноты тестирования нашего входного набора потребуется 10 миллиардов проверок! Сравните, например, с микросхемой дешифратора, преобразующего входное 4-разрядное двоичное значение в сигнал на одном из 16 выходов. Входных наборов будет всего 16, а таблица истинности состоит из $16^2 = 256$ значений.

На практике программист применит допустимую эвристику и будет тестировать, например, только несколько значений (один миллион, ноль, случайная величина из диапазона) для нескольких типовых конвертаций из 100 возможных, дополнительно проверяя допустимую точность значений на входе. При этом формальный показатель покрытия модульными тестами по-прежнему будет 100 %...

Но это ещё не всё.

Микросхема работает с заданной тактовой частотой. Если, например, частота равна 1 МГц, то подав на вход набор значений, вы гарантированно через одну микросекунду получите результат на выходе.

Если же вы подадите набор значений на вход нашего компонента, то время отклика будет неопределённым. Может быть, программа отработает за секунду.

Может быть, зависнет навечно, если не предусмотрен тайм-аут. А если несколько параллельных запросов?

Поэтому вдобавок к модульному тесту необходимо программировать тест производительности (нагрузочный), который тем не менее *не гарантирует* время отклика, а только позволяет определить его *ожидаемое значение* при некоторых условиях.

Таким образом, собрав из кучи микросхем устройство, мы уверены, что оно будет работать:

- согласно таблицам истинности;
- с заданной тактовой частотой.

Собрав же из компонентов программу, мы можем только:

- приблизительно и с некоторой вероятностью оценивать время отклика на выходе;
- в большинстве случаев ограничиться выборочным тестированием, забыв о полноте.

Если вам говорят: «Пришло время собранных из кубиков программ», будьте в курсе ограничений технологии. Очень уж далеки программные компоненты от электронных кубиков.

Безысходное программирование

Любая программа, даже созданная визуально, имеет в своей основе исходный код на каком-либо языке программирования.

Безысходное программирование — это программирование без «исходников». То есть мы пишем свой код, не имея исходных текстов используемой подпрограммы, класса, компонента и т. п.

Когда необходимо обеспечить гарантированную работу приложения, включающего в себя сторонние библиотеки или компоненты, то, не имея доступа к их исходному коду, вы остаётесь один на один с «чёрным ящиком». Даже покрыв их тестами, близкими к параноидальным, вы не сможете понять всю

внутреннюю логику работы и предусмотреть адекватную реакцию системы на нестандартные ситуации. Поэтому программирование без исходников в таком сценарии превращается в настоящую безысходность и безнадёгу.

Пока цена ошибки в приложении — потеря нескольких строк введённой пользователем информации, дело может ограничиться долгоживущей записью в базе данных ошибок, закрываемой не её исправлением, а описанием обхода «граблей»¹. Но ситуация кардинально поменяется, если цена будет исчисляться многими нулями потерь от упущенной сделки в торговой системе, сотнями исков клиентов, получивших неверные счета, или того хуже — аварией на производстве. Ответственность с разработчиков никто не снимал.

В рамках аудита нередко приходилось наблюдать, как правят программный код триггеров и хранимых процедур прямо в базе данных. Ассоциация с этим непотребством у меня тесно связана с утилитой `debug`, которая в MS DOS позволяла писать машинные команды прямо в память. Или с командой `type > program.com` для набора машинного кода с консоли в исполняемый файл. Понятное дело, что занимаются такими вещами при разработке программного обеспечения только от безысходности.

Частным, но частым случаем безысходного программирования является софтостроение без использования системы управления исходным кодом (*revision control system*), позволяющей архивировать и отслеживать все его изменения.

Эволюция аппаратуры и скорость разработки

В 1980-х годах у японцев существовала программа по созданию ЭВМ 5-го поколения. К сожалению, цель достигнута не была, хотя проявилось множество побочных эффектов вроде всплеска интереса к искусственному интеллекту, популяризации языка Пролог, да и отрицательный опыт — тоже опыт, возможно, не менее ценный.

В итоге, спустя 20 с лишним лет, все мы — и разработчики, и пользователи — продолжаем сидеть на «числогрызах» 4-го поколения. Производительность «железа» возросла на порядки, почти упёршись в физические ограничения миниатюризации полупроводников и скорость света. Стоимость тоже на порядки, но снизилась. Увеличилась надёжность, развилась инфраструктура, особенно

¹ Грабли — синоним скрытой ошибки в программе. «Наступить на грабли» в программистском фольклоре означает выявить скрытую проблему за собственный счёт.

сетевая. Параллелизация вычислений пошла в массы на плечах многоядерных процессоров.

Прежними остались лишь принципы, заложенные ещё в 1930-х годах и названные, согласно месту, Гарвардской и Принстонской архитектурами ЭВМ. Вчерашний студент теперь пишет не на ассемблере и С, а на Java, будучи уверенным в принципиальной новизне ситуации, не всегда осознавая, что изменилось только количество герц тактовой частоты и байтов запоминающих устройств.

Взросла ли при этом скорость разработки?

Вопрос достаточно сложный, даже если сузить периметр до программирования согласно постановке задачи. Тем не менее я рискнул бы утверждать, что не только не возросла, но, наоборот, снизилась.

Массовые технологии, доступные шести миллионам программистов, являются универсальными, то есть могут быть использованы для разработки большинства типов программ, пакетов и систем. Поэтому важным элементом бизнеса становится не столько сокращение срока разработки, сколько максимизация использования стандартных сред, компонентов и фреймворков. И хотя по срокам, бюджету и количеству разработчиков владеющие специализированными технологиями выигрывают у бригады «универсалов», но возникающие при этом риски могут свести к минимуму весь выигрыш.

Конечно, специализированные средства разработки всегда обеспечат преимущества по сравнению с универсальными. Тем не менее основная разработка по-прежнему будет идти на весьма ограниченном наборе универсальных сред и фреймворков, выталкивая специализированную в ниши, где сроки и производительность являются наиболее важными.

Бывшие разработчики PowerBuilder или FoxPro неоднократно выражали мне своё недоумение по поводу того, что для простейших операций, вроде настраиваемого показа табличных наборов данных на клиенте, теперь приходится тратить уйму времени и писать десятки строк кода, а каждая корректировка структур данных должна быть отражена во всех слоях системы. Опуская технические ограничения классических клиент-серверных приложений, нетрудно убедиться, что найти на рынке труда специалиста по PowerBuilder на порядок сложнее, чем VB.NET-программиста. К тому же поставщик среды PowerBuilder после многих перекупок за последние годы в итоге выглядит не слишком жизнеспособным.

С другой стороны, количество работающих в софтверной индустрии женщин росло до начала 1990-х годов, после чего резко пошло на убыль. Рисунок 2 представляет ситуацию в США, но и в СССР и позднее в РФ она вряд ли отличалась.

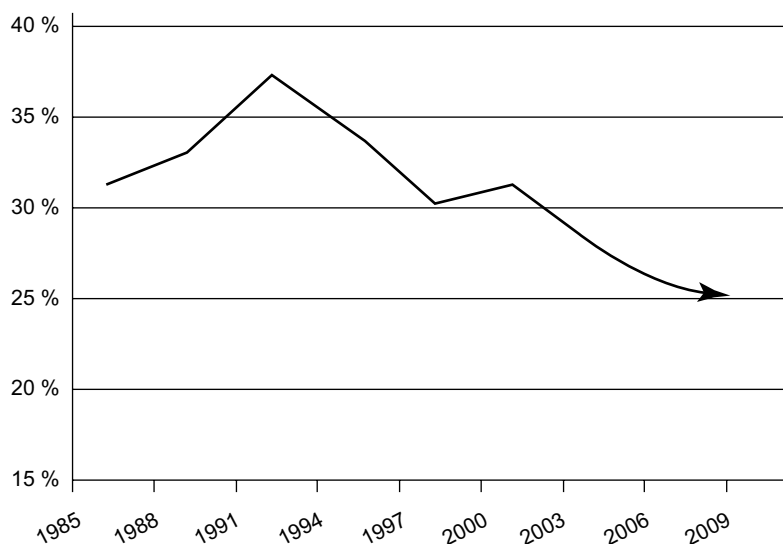


Рис. 2. Процент женщин, занятых в компьютерной отрасли в 1985–2009 годах, согласно данным американского бюро статистики труда

Такая тенденция иллюстрирует факт ухода технологий софтверного строительства от специализированных сред, не требующих работы на далёком от решаемой прикладной задачи уровне математических абстракций, в которых прекрасный пол почему-то считается менее способным разбираться.

В начале своей трудовой деятельности я наблюдал изображённый на графике пик в среде женщин-программистов. В отделах конструкторско-технологического центра профессия прикладного программиста прекрасно сочеталась с равноправием: мужчин и женщин среди них было примерно поровну. В разгаре был переход с больших ЭВМ на «персоналки» и локальные сети NetWare. Автоматизированные информационные системы переносили на новую платформу, используя специализированные среды разработки приложений баз данных типа FoxPro, Clipper, dBase. И женская половина коллектива успешно справлялась с поставленными перед ними задачами.

Одна умная девушка, получавшая в школе хорошие отметки по математике и без особого труда оперирующая программами и данными в среде FoxPro, после знакомства со средой C++ Builder высказалась максимально ясно: «Язык понравился, но я не поняла, зачем мне нужны эти классы...».

Впоследствии мне не раз приходилось видеть исходники программисток на вполне себе объектно-ориентированном Delphi/C++Builder. В прикладном коде никакого объектного подхода, конечно, не было, всё ограничивалось компоновкой экранных форм стандартными элементами среды и написанием обработчиков событий в процедурном стиле.

Разумеется, это говорит не о «плохости» ООП, а о высоком уровне компетенции, необходимом, чтобы эта технология давала осязаемые преимущества. Тогда как цель прикладника — побыстрее собрать работающее решение для заказчика. Неплохим компромиссом был Visual Basic, похороненный Microsoft в начале 2000-х годов. Хотя ему и далеко до специализированных сред по удобству и скорости, VB не навязывал следование ООП, давая органично встраивать процедурную обработку между склеенными компонентами.

Про объектно-ориентированный подход мы ещё поговорим, но у меня сложилось мнение, что будучи реализованной повсеместно без малейших представлений о её применимости, эта технология сыграла не последнюю роль в вытеснении женского труда из отрасли.

Диалог о производительности

В одном из проектов у меня произошёл с заказчиком весьма характерный разговор, ярко иллюстрирующий приоритеты при освоении бюджета даже в относительно небольшой частной компании:

Заказчик: Нам необходимо рассчитать ряд показателей на основе данных одной базы, но использовать их будут из таблиц в другой базе данных.

Я: Сделаем расчёт на SQL, заполним таблицы напрямую. Базы данных у вас на одном сервере?

Заказчик: На одном, но теоретически могут быть разнесены...

Я: Значит, просто поменяется источник расчётных данных: локальный на удалённый.

Заказчик: Э-э-э... А по сравнению с пакетом сервиса интеграции¹ скорость не замедлится?

Я: Наоборот, все будет работать быстрее. Две стадии — запрос с расчётами и заливка результата — вместо трёх.

Заказчик: Ух ты, здорово! (*мнётся*)

Я (*с пониманием в голосе*): Если вы хотите привлечь к работе ещё одного человека, то мы заполним данные в расчётной базе, а потом ваш сотрудник сделает пакет, который просто перекачает данные из одной базы в другую.

Заказчик (*радостно*): Да, я бы предпочёл сделать так!

Речь шла о регулярном заполнении пары таблиц объёмом примерно в сотню миллионов строк. Вместо прямого пути с настраиваемым источником данных ради приобщения к действию ещё одного «выпускника курсов» заказчик выбрал локальный расчёт с последующий перекачкой данных. В итоге используемое дисковое пространство удваивается, время увеличивается.

Служба «бизнес-интеллекта»² предприятия — неисчерпаемый кладёз такого рода задач для новоиспечённых специалистов курсов переквалификации и их начальников.

О карманных монстрах

В послужном списке одной конторы имелась небольшая и простая система ведения заказов на рекламу для книжно-журнального издательства. Полтора десятка сущностей (и таблиц), с десятков экранных форм.

Лет 10–15 назад можно было бы взять на выбор Delphi/C++ Builder, PowerBuilder, Visual Basic, FoxPro, лёгкую клиент-серверную СУБД и сделать

¹ SSIS — SQL Server Integration Services, служба управления пакетами обработки данных, внешних по отношению к СУБД. По сути, среда для быстрой визуально-скриптовой разработки программ-конвертеров данных.

² От английского «Business Intelligence» — служба предприятия, занятая аналитической обработкой данных.

приложение за 3–5 дней с написанием каких-то сотен строк прикладного кода. Внесение изменений типа «добавления атрибута к сущности» вместе с воссозданием инсталлятора и скрипта обновления базы данных занимало час-два.

В 2009 году приложение было сделано на платформе .NET в трёхзвенной архитектуре: сервер приложений на базе WCF, Entity Framework, СУБД SQL Server 2005 и клиент в виде подключаемого модуля (*add-in*) к Office 2007 на WinForms. Спасибо, что не на WPF. Приложение занимает примерно 20 тысяч строк на C#, из них более половины являются техническими: слой объектов доступа к данным, прокси классов для WCF и прочая начинка. Конфигурационный файл для WCF-сервера — 300 строк XML. Это больше, чем нужно написать, например, Delphi-кода для логики отображения форм во всем приложении.

Первоначальная разработка заняла у фирмы порядка трёх недель работы одного программиста при том, что большая часть кода генерируется из модели. Отладка проблемы в канале WCF при нештатном исключении занимает часы. При добавлении атрибута изменение поднимается по всем звеньям, что также может потребовать длительного времени.

Наверное, и в 2009 году можно было бы обойтись разработкой на VB.NET приложения, напрямую работающего с СУБД через DataSet. И даже с учётом необходимости устанавливать .NET на рабочем месте, это было бы не намного хуже и медленнее, чем 10–15 лет назад.

Но механизм принятия решения базировался на других критериях:

- менеджеру, в соответствии с корпоративным стандартом, необходимо было использовать только платформы и средства Microsoft;
- программист не имел опыта разработки вне шаблонов многозвенной архитектуры и проекций объектов на реляционную СУБД, поэтому не стал рисковать.

Такие решения принимаются в мире ежечасно. Поэтому новичкам не раз предстоит столкнуться с заданием типа «быстро добавить поле в форму» и познакомиться с внутренним устройством подобных программ — карманных монстров, готовых откусить палец неосторожно сунутой руки.

ASP.NET и браузеры

Всякий раз, когда приходилось что-то делать при помощи технологии ASP.NET или просто править чей-то код, даже правильно написанный, меня не покидало ощущение копания по локоть в большой столовской кастрюле с макаронами.

Давайте вспомним историю. Успех в 1994–1995 годах первой версии бесплатной и открытой платформы PHP, называвшейся тогда Personal Home Page, показал, что веб быстрым темпом трансформируется из источника статической информации в среду динамических интерактивных приложений, доступных через «стандартный» проводник-браузер. Ниже я объясню, почему взял слово «стандартный» в кавычки. Microsoft не могла остаться в стороне и выдала собственное решение под названием ASP (Active Server Pages), работающее, разумеется, только под Windows.

Лежащий в основе названных платформ принцип был просто замечательным, хотя и совсем не новым. Логика приложений реализовывалась на стороне сервера скриптами на интерпретируемом языке, тонкий клиент-браузер в качестве терминала только отображал информацию и ограниченный набор элементов управления вроде кнопок. Вскоре выяснилось, что привыкшему к интерактивности полноценных приложений пользователю одних лишь кнопок не хватает. Тогда и в браузеры (то есть на стороне клиента) тоже включили поддержку скриптовых языков.

В итоге исходная веб-страница, ранее содержавшая только разметку гипертекста, стала включать в себя скрипты для выполнения вначале на сервере, а затем и на клиенте. Можете представить, какова была эта «лапша» на скольконибудь сложной странице ASP. Многие сотни строк каши из HTML, VBScript и клиентского JavaScript.

Последующая эволюция технологии была посвящена борьбе с этой лапшой, чтобы программный код мог развиваться и поддерживаться в большем объёме и не только его непосредственными авторами. На другом фронте бои шли за отделение данных от их представления на страницах, чтобы красивую обёртку рисовали профессиональные дизайнеры-графики, не являющиеся программистами.

Однако, несмотря на значительный прогресс за последние 15 лет, производительность разработки пользовательского интерфейса для веб-приложений в разы отстаёт от автономных приложений, тех самых, что «компонентокидали» на Visual Basic, Delphi или C++ Builder делали 15 лет назад.

Если взять простой пример отображения модального диалога, то в Delphi, Visual Basic или WinForms-приложении потребуется написать одну строку кода для вызова формы и вторую — для проверки статуса возврата. Для веб-приложения, во-первых, реализация этого сценария одними серверными скриптами невозможна, необходимо задействовать клиентские. Во-вторых, необходимо хорошо представлять себе механизмы взаимодействия браузера и веб-сервера, чтобы синхронизировать вызовы и организовать передачу статуса. Наконец, веб-приложение не имеет состояния, поэтому понятие пользовательской сессии очень условное. Например, после 15-минутной паузы в деятельности клиента сервер решает, что сеанс закончен.

Теперь представьте, что под модальным окном с индикатором выполнения и единственной кнопкой «Прервать» вам надо запустить асинхронную обработку с обновлением информации в главном окне. В автономном приложении снова пишем несколько строк кода, добавляя обработчик события с делегатом из основной формы. А вот в веб... Даже краткое описание займёт несколько абзацев и будет касаться зоопарка технических ухищрений.

В качестве иллюстрации, существующая подсистема пользовательского интерфейса у одного из наших клиентов насчитывала всего около четырёх десятков экранных форм. Но для реализации только логики отображения потребовалась примерно сотня тысяч (!) строк *code-behind*¹ и Java-скриптов, несмотря на то, что создатели чётко отделили слой представлений от прикладной обработки, следовали логике «модель—представление—контроллер»², а общие элементы управления разного уровня — от собственных (*custom*) до композитных (*user*) — свели в библиотеки.

Легко проследить даже на простом примере, что для программиста помимо решения собственно прикладной задачи находится уйма забот. Основной целью такой дополнительной головной боли является платформенная независимость клиентской части приложения и максимально облегчённое развёртывание так называемого «тонкого» клиента, которым является веб-браузер.

Действительно, переносить автономное приложение между разными операционными системами и аппаратными платформами трудно. Большинство из

¹ Возможность разделения визуальной части и бизнес-логики по разным файлам. Одна из ключевых концепций в ASP.NET.

² От англ. «MVC, Model-View-Controller» — концепция построения приложений графического интерфейса пользователя. Развитие концепции, полностью исключающее связь модели и вида — MVP, Model-View-Presenter.

них пишутся под Windows. Приложения на Java или WinForms .NET переносить легче, но для развёртывания требуется предустановленная среда времени исполнения (*runtime*) соответствующего фреймворка не ниже определённой версии. Гораздо меньше проблем с развёртыванием у FreePascal/Lazarus (открытый многоплатформенный аналог Delphi), новой версии Delphi XE или C++/Qt-приложений. Но, во-первых, перечисленное — далеко не самые массовые технологии, представляющие по этой причине дополнительные риски для менеджеров. Во-вторых, для обеспечения переноса и сам код, и требования к его написанию усложнятся, тогда как тестировать придётся на всех целевых платформах.

Поэтому на первый взгляд идея универсального программируемого терминала, которым является веб-браузер, поддерживающий стандарты взаимодействия с веб-сервером, выглядит привлекательно. Никакого развёртывания, никакого администрирования на рабочем месте. Именно этот аргумент и стал решающим в конце 1990-х годов для внедрения веб в корпоративную среду. Гладко было на бумаге, но забыли про овраги...

Достаточно быстро выяснилось, что разработка приложения, корректно работающего хотя бы под двумя типами браузеров (Internet Explorer, Netscape и впоследствии Mozilla) — задача не менее сложная, чем написание кода в автономном приложении на базе переносимой оконной подсистемы (Lazarus, C++ и другие). А тестировать нужно не только под разными браузерами, но и под разными операционными системами. С учётом версий браузеров.

Поскольку отступать было поздно (см. информацию про капиталовложения в начале раздела), эту проблему решили в лоб. Корпоративная среда в отличие от общедоступного Интернета имеет свои стандарты. Поэтому при разработке веб-приложений достаточно было согласовать внутренние требования предприятия с возможностями разработчиков. К началу 2000-х годов установился фактический стандарт корпоративного веб-приложения: Internet Explorer 6 с последним пакетом обновления под Windows 2000 или Windows XP.

Под эти требования за 10 лет было написано великое множество приложений. А когда пришла пора обновлять браузеры, внезапно выяснилось, что их новые версии далеко не всегда совместимы с находящимися в эксплуатации системами. И по этой причине простое обновление Internet Explorer 6 на 7 вызовет паралич информационных систем предприятия.

Достаточно свежий пример. В одной крупной конторе (более 10 тысяч сотрудников) система учёта рабочего времени из Internet Explorer 7, 8 или 9 на основной странице ввода закидывалась, эмулируя скриптами щелчки мыши

и подвешивая браузер. В Firefox 3 заикливания не происходило, но не работали всплывающие окна. В более поздних версиях Firefox система не работала совсем, выдавая «*browser is not supported*». В Chrome корректно работала предварительная версия, но сданная в эксплуатацию почему-то лишилась этого качества с выдачей сообщения о несовместимости: «*The iView is not compatible with your browser, operating system, or device*».

Итого в 2011 году приложение по-прежнему стабильно работало *только в Internet Explorer 6*, выпущенном в 2001 году, то есть 10 лет назад.

За эти же 10 лет прошло огромное количество презентаций о том, что инвестиции сделаны не зря, о том, как замечательно работают веб-приложения в интранете и корпоративной среде в целом. На деле же оказалось, что, собрав свои приложения на базе веб-технологии, корпорация оказалась заложником версии и марки конкретного браузера. Даже переход с Internet Explorer 6 на 7, не говоря уже о Firefox или Chrome, оказался катастрофой масштаба предприятия с долгими месяцами миграции и последующей стабилизации. Разумеется, если есть кому стабилизировать, ведь за 5–10 лет сменяются разработчики, уходят с рынка прежние поставщики. Для таких случаев приложение остаётся жить на виртуальной машине под старой версией операционной системы и проводника.

Предприятие оказывается один на один с веб-технологией, которая, как утверждали вначале, ничего не стоит при развёртывании и не требует администрирования на рабочем месте. Про затраты на обновление браузера, конечно, тогда никто не заикался, хотя соблюдения в необходимом объёме стандартов веб-терминалами как не было, так и нет.

Менеджеры другой фирмы стали искать решение проблемы у Google, отдав ему на откуп корпоративный документооборот, групповую работу и почту. Новое обоснование выглядело так: «Уж Google-то обеспечит совместимость приложений со своим браузером!» Не знаю, слышали ли поверившие в такой довод хоть что-нибудь об открытых системах и о печально завершившейся истории с IBM, продававшей свои большие ЭВМ вместе со своим же, привязанным к ним программным обеспечением. Человеческая история имеет свойство повторяться.

Вернемся к классическим автономным приложениям: даже в самом примитивном варианте развёртывания при использовании обычных исполняемых файлов с запуском с разделяемого сетевого диска обновление одной программы не вызывает крах остальных. Не зря тот же SAP для работы с R/3, ключевой системой предприятия, использует самое что ни на есть полноценное оконное приложение, оставляя веб для частных случаев.

Факт наличия у большинства корпоративных клиентов только Internet Explorer версии 6 в качестве стандарта не раз оборачивался казусами. Так, обновление нашего внутреннего сервера Microsoft Exchange привело к тому, что веб-почта в Internet Explorer 6 стала работать со множеством ограничений, например, отсутствует разметка текста, нет проверки орфографии, не показывается дерево папок. Чтобы отправить почту, пришлось соединяться с сервером (!), где изначально стоял Internet Explorer 7, и работать оттуда через терминал.

Напоследок хочется пожелать коллегам, ответственным за выбор технологий, всячески обосновывать необходимость использования веб-интерфейса в вашей системе, принимая в рассмотрение другие пути.

Апплеты, Flash и Silverlight

Появившись в 1995 году, технология Java сразу пошла на штурм рабочих мест и персональных компьютеров пользователей в локальных и глобальных сетях. Наступление проводилось в двух направлениях: полноценные «настольные» (*desktop*) приложения и так называемые апплеты¹, то есть приложения, имеющие ограничения среды исполнения типа «песочница» (*sandbox*). Например, апплет не мог обращаться к дискам компьютера.

Несмотря на значительные маркетинговые усилия корпорации Sun, результаты к концу 1990-х годов оказались неутешительны: на основной платформе пользователей — персональных компьютерах — среда исполнения Java была редким гостем, сами приложения можно было сосчитать по пальцам одной руки (навскидку вспоминается только Star Office), веб-сайтов, поддерживавших апплеты, было исчезающе мало, а настойчивые просьбы с их страниц скачать и установить 20 мегабайтов исполняемого кода для просмотра информации выглядели издевательством при существовавших тогда скоростях и ограничениях трафика. Несомненно, судебная тяжба Sun в 1997 году с Microsoft, тут же прекратившей распространение Java вместе с Windows, также сыграла свою роль. Но основными объективными причинами такого исхода были:

- универсальность и кроссплатформенность среды, обернувшаяся низким быстродействием и невыразительными средствами отображения

¹ От английского термина «applet» — приложеньце.

под вполне конкретной и основной для пользователя операционной системой Windows;

- необходимость установки и обновления среды времени исполнения (Java runtime).

В 2007 году Sun утверждала, что среда исполнения Java установлена на 700 миллионах персональных компьютеров¹, правда не уточнялась её версия. В декабре 2011 года уже новый владелец — корпорация Oracle — привёл данные о том, что Java установлена на 850 миллионах персональных компьютеров и миллиардах устройств в мире². Но поезд ушёл, развитие приложений на десктопах сместилось далеко в сторону по пути начинённых скриптами веб-браузеров, а рост количества мобильных устройств положил конец монополии «персоналок» в роли основного пользовательского терминала.

Тем не менее необходимость в кросс-платформенных богатых интерактивными возможностями интернет-приложениях³ никуда не исчезла, поскольку браузеры, нашпигованные скриптовой начинкой, обладали ещё большими техническими ограничениями и низким быстродействием даже по сравнению с апплетами. Эта ниша к началу 2000-х годов оказалась плотно занятой Flash-приложениями, специализирующимися на отображении мультимедийного содержания. Учтя ошибки Java, разработчики из Macromedia сделали установку среды исполнения максимально лёгкой в загрузке и простой в установке.

Упомянутая специализация технологии на интерактивном мультимедийном содержании веб-сайтов, включая потоковое аудио и видео, с другой стороны, оказалась непригодной для использования в разработке корпоративных приложений, продолжавшей по этой причине использовать браузеры со скриптами.

К решению проблемы подключилась Microsoft. Первым «блином» в 2005 году стала технология ClickOnce развёртывания полноценных WinForms-приложений. По-прежнему клиентское рабочее место требовало предварительно установки среды исполнения .NET версии 2. Но развёртывание и автоматическое обновление приложения и его компонентов было полностью автоматизировано. Первоначально пользователь, не имеющий прав локального администратора,

¹ «Java Runtime Environment is found on over 700 million personal computers», пресс-релиз Sun, 2007.

² «Java runs on more than 850 million personal computers worldwide, and on billions of devices worldwide», пресс-релиз Oracle, 2011.

³ От англ. RIA — Rich Internet Applications.

устанавливал необходимую программу, просто щёлкнув по ссылке в браузере, далее запуская её с рабочего стола или из меню. Sun отреагировала молниеносно, добавив аналогичную возможность под названием Java Web Start.

Но «блин» всё-таки вышел комом. По данным AssetMetrix¹, основной парк корпоративных компьютеров в 2005 году составляли «персоналки» под управлением Windows 2000 (48 %) и Windows XP (38 %). Имея полную возможность предустановить среду .NET 2 на все эти рабочие места вместе с очередным пакетом обновлений, Microsoft не решилась на такой шаг, тем самым фактически похоронив массовое использование новой технологии разработчиками, имевшими неосторожность надеяться на помощь корпорации в развёртывании тяжёлых клиентских приложений.

Возможно, одной из причин стала потеря интереса Microsoft к WinForms, чьё развитие было заморожено, и переход в .NET 3 к более общей технологии построения пользовательских интерфейсов WPF², отличающейся универсальностью и большей трудоёмкостью в прикладной разработке, но позволяющей полностью разделить труд программистов и дизайнеров, что имело смысл в достаточно больших и специализированных проектах. Вот вам очередная иллюстрация к теме прогресса в производительности разработки.

Побочным продуктом WPF стал Silverlight. По сути, это реинкарнация Java-апплетов, но в 2007 году, спустя более 10 лет, и в среде .NET. Кроме того Silverlight должен был по замыслу авторов составить конкуренцию Flash в области мультимедийных интернет-приложений.

В отличие от WPF, Silverlight вызвал больший энтузиазм разработчиков корпоративных приложений. Во-первых, для развёртывания не требовалась вся среда .NET целиком, достаточно было установить её часть, размер дистрибутива которой составлял всего порядка 5 мегабайтов. Поэтому на очередные обещания Microsoft предустановить .NET 3 можно было не полагаться, тем более при уже анонсированном .NET 3.5. Во-вторых, приложение можно было запускать не только в окне браузера, но и автономно.

Наша контора среагировала достаточно быстро, и к 2009 году в софто-строительной фабрике уже имелся номинальный генератор кода по модели для Silverlight-приложений. Ожидая взросления и стабилизации технологии,

¹ «In Q1 2005 48% of business PCs ran Windows 2000, 38% ran Windows XP», исследование AssetMetrix, 2005.

² Windows Presentation Foundation — технология Microsoft построения Windows-приложений с богатыми возможностями отображения информации и графики.

периодически подступаясь к теме, я собирал мнения коллег о встретившихся им подводных камнях.

Прежде всего насторожили меня новости про отсутствие в Silverlight отличных от юникода¹ кодировок. Их нет в константах, а `Encoding.GetEncoding (1251)` выдаёт ошибку. Как корректно импортировать в приложение ASCII²-файл? Никак. Из этого вытекала невозможность полноценной работы приложения с обыкновенным текстовым файлом данных, вроде CSV (*comma separated values*).

Прямой доступ к базам данных также отсутствовал. Можно было пойти окольными путями через COM interops и ADO, но для этого требовались очень серьёзные поводы.

И тут в корпорации, аккурат к октябрьской конференции разработчиков 2010 года, издали новый декрет: «Наша стратегия по Silverlight изменилась»³. Сессий по новой версии Silverlight 5 на мероприятии не было вовсе. Снова часы пробили полночь, и карета превратилась в тыкву. Приоритетом стал HTML 5.

Silverlight вырос до вполне взрослой версии 4, уже давно вышла Visual Studio 2010, где встроена поддержка разработки приложений под него. Но зададимся вопросом: «Может ли пользователь установить себе Silverlight-приложение, не будучи администратором на своем компьютере?» Ответ, мягко говоря, разочаровывающий: «Нет, не может».

Это значит, что развёртывать Silverlight-песочницы на машинах пользователей должны сами компании через своих специалистов, ответственных за инфраструктуру. Хотя в соответствующем официальном документе описано много способов облегчения администраторской деятельности, факт остаётся фактом: технология в своей 4-й (!) версии не может быть использована в корпоративной среде без серьёзных накладных расходов.

Итак, итог к 2012 году. Во-первых, «старые» технологии вроде автономного оконного кроссплатформенного приложения на Lazarus/FreePascal, Delphi XE или Qt/C++ по-прежнему позволяют сделать то, что нельзя сделать «новыми и прогрессивными». Во-вторых, ценность Silverlight по сравнению с полноценным .NET на уровне развёртывания практически нулевая. Видимо, по этой причине Microsoft недавно закрыла веб-сайт silverlight.net, в очередной раз оставив разработчиков в интересном положении.

¹ От англ. unicode — международный стандарт кодирования символов, позволяющий представить знаки практически всех известных алфавитов, включая иероглифы.

² American Standard Code for Information Interchange — американская стандартная таблица кодов печатных символов.

³ См. пресс-релиз компании Microsoft «Our strategy with Silverlight has shifted».

Из продвигаемых Microsoft за последние 10 лет технологий для разработки полноценных пользовательских интерфейсов, не заброшенных на пыльный чердак, остался только WPF, имеющий весьма сомнительную ценность для небольших коллективов и отдельных разработчиков. WPF — это ниша крупных автономных Windows-приложений. Кроме того, сама по себе она невелика, в ней уже есть WinForms — более простой и быстрый в разработке фреймворк, к тому же переносимый под Linux/Mono. Поэтому при соответствующих ограничениях развёртывания выбор по-прежнему лежит между веб-браузером или условным Delphi, хочешь ты этого или нет...

ООП — неизменно стабильный результат

Говоря об объектно-ориентированном подходе и программировании, принято добрым словом вспоминать начало 1970-х годов и язык Smalltalk, скромно умалчивая, что понадобилось ещё почти 15 лет до начала массового применения технологии в отрасли, прежде всего, за счёт появления C++ и позднее — Объектного Паскаля. Потому что фактическим отраслевым стандартом был язык С, а Паскаль широко использовался для обучения и в основном для прикладного программирования, если не рассматривать исключения вроде первой версии Microsoft Windows. Религиозные войны 1970–80-х годов в новостных группах проходили под лозунгом «Си против Паскаля». По этой причине революционный переход сообществ на Smalltalk выглядел маловероятным, тогда как объектно-ориентированные расширения вышеупомянутых языков были восприняты положительно. Немудрено, что многие концепции Smalltalk были в них реализованы.

В начале широкой популяризации ООП, происходившей в основном за счёт языка C++, одним из главных доводов был следующий: «ООП позволяет увеличить количество кода, которое может написать и сопровождать один среднестатистический программист». Приводились даже цифры, что-то около 15 тысяч строк в процедурно-модульном стиле¹ и порядка 25 тысяч строк на C++.

Довод в целом правильный, хотя из него совсем не следовало, что десяти программистам на C++ будет легче сопровождать общую систему, чем десяти программистам на С. Про это как-то забыли, потому что существовало много автономных проектов, управляемых процессом типа бруксовской операционной

¹ В языке С нет понятия модуля, поэтому этот показатель несколько ниже.

бригады¹ с главным программистом, отвечающим за всё решение. Собственно, и Бьёрн Страуструп, создатель C++, прежде всего преследовал цели увеличения производительности своего программистского труда.

Как только «главным программистом» стал «коллективный разум» муравейника, неважно мечущийся ли на планёрках «гибкой» (*agile*) разработки, про заседавший ли на совещаниях по тяжёлой поступи RUP², проблема мгновенно всплыла, порождая Ад Паттернов³, Чистилище нескончаемого рефакторинга⁴ и модульных тестов, недостижимый Рай генерации по моделям кода безлюдного Ада.

Термин «Ад Паттернов» может показаться вам незнакомым, поэтому я расшифрую подробнее это широко распространившееся явление:

- слепое и зачастую вынужденное следование шаблонным решениям;
- глубокие иерархии наследования реализации, интерфейсов и вложения при отсутствии даже не очень глубокого анализа предметной области;
- вынужденное использование все более сложных и многоуровневых конструкций в стиле «новый адаптер вызывает старый» по мере так называемого эволюционного развития системы;
- лоскутная⁵ интеграция существующих систем и создание поверх них новых слоёв API⁶.

¹ Ф. Брукс описывает софтостроение по принципу «операционной бригады» в своей книге «Мифический человеко-месяц»[0].

² Rational Unified Process — итеративная тяжеловесная методология софтостроения от компаний Rational и IBM.

³ От англ. design pattern — шаблон проектирования.

⁴ От англ. refactoring — реструктуризация и факторизация программного кода. В экстремальных методиках при отсутствии концепции системы и анализа предметной области формально требуется постоянный рефакторинг кода, при помощи которого предполагается чудесным образом прийти к хорошему решению ничего не проектируя.

⁵ Термин широко используется в автоматизации предприятий и происходит от «лоскутного одеяла» — разрозненного набора программ и пакетов, решающих локальные задачи подразделений.

⁶ API (Application Programming Interface) — интерфейс программирования приложений, функциональность, которую предоставляет модуль, компонент или библиотека программисту.

В результате эволюционного создания Ада Паттернов основной ценностью программиста становится знание, как в данной конкретной системе реализовать даже простую новую функцию, не прибегая к многодневным археологическим раскопкам и минимизируя риски дестабилизации. Код начинает изобиловать плохо читаемыми и небезопасными конструкциями:

```
Services.Oragnization.ContainerProvider.ProviderInventory.InventorySectorPrivate.  
Stacks[0].Code.Equals("501")
```

Последствия от создания Ада Паттернов ужасны не столько невозможностью быстро разобраться в чужом коде, сколько наличием многих неявных зависимостей. Например, в рамках относительно автономного проекта мне пришлось интегрироваться с общим для нескольких групп фреймворком ради вызова единственной функции авторизации пользователя: передаёшь ей имя и пароль, в ответ «да/нет». Этот вызов повлёк за собой необходимость явного включения в .NET-приложение пяти сборок. После компиляции эти пять сборок притащили за собой ещё более 30, большая часть из которых обладала совершенно не относящимися к безопасности названиями, вроде XsltTransform. В результате объём дистрибутива для развёртывания вырос ещё на сотню мегабайтов и почти на 40 файлов. Вот тебе и вызвал функцию...

Разумеется, превратить код программы в тарелку спагетти можно без особого труда и в процедурно-модульной технологии. Разница в том, что распутывать процедурное спагетти гораздо легче, чем лапшу объектно-ориентированную. Потому что кроме вложенности вызовов процедур в ООП имеет место различного типа вложенность объектов — от наследования реализации до многоуровневых ассоциаций, и совмещение в классах собственно структур данных и процедурного кода.

Несомненно, C++ является мощным инструментом программиста, хотя и с достаточно высоким порогом входа, предоставляющим практически неограниченные возможности профессионалам с потребностью технического творчества. Я видел немалое количество примеров изящных фреймворков и прочих «башен из слоновой кости», выполненных одиночками или небольшим коллективом. Но крупные проекты подвержены влиянию уже упоминавшейся гауссианы (см. рис. 1). Нормальное распределение вовлекает в процесс большое количество крепких профессионалов-середняков, которым надо сделать «чтобы работало» с наименьшими телодвижениями во время нормированного рабочего дня. Если Microsoft или Lockheed Martin — подрядчик Министерства Обороны США, имеют возможность растянуть кривую на графике вправо и вложить немалые

средства во внутреннюю стандартизацию кодирования, то в обычной ситуации оказывается, что C++, действительно увеличивавший личную продуктивность Страуструпа и его коллег, начинает тормозить производительность большого софтостроительного цеха где-нибудь в жарком субтропическом опенспейсе¹ площадью в гектар. Помимо общих проблем интеграции, на C++ достаточно просто «выстрелить себе в ногу», и человеческий фактор быстро становится ключевым риском проекта.

Если вернуться к вопросу стандартов кодирования на C++, хорошим примером будет разработка программной начинки нового истребителя F-35 [15]. Объем разработанного кода порядка 10 миллионов строк, это даже больше, чем Windows. Следовательно, стандарт вполне пригоден к практическому использованию. Но имеются ли у вас в проекте ресурсы для того, чтобы не только обучить всех программистов 150-страничному своду правил, но и постоянно контролировать его исполнение?

Поэтому появились новые C-подобные языки: сначала Java, а чуть позже и C#. Они резко снизили порог входа за счёт увеличения безопасности программирования, ранее связанной прежде всего с ручным управлением памятью. Среды времени исполнения Java и .NET решили проблему двоичной совместимости и повторного использования компонентов системы, написанных на разных языках для различных аппаратных платформ.

Когда многие технические проблемы были решены, оказалось, что ООП очень требовательно к проектированию, так и оставшемуся сложным и недостаточно формализуемым процессом. Похожая ситуация была в середине XX века в медицине: после изобретения антибиотиков первое место по смертности перешло от инфекционных болезней к сердечно-сосудистым.

Примерно в то же время в сообществе начались дискуссии, появились первые публикации вроде уже ставшей классической «Почему объектно-ориентированное программирование провалилось?»². Эксперты по ООП в своих книгах стали нехотя писать о том, что технология тем эффективнее, чем более идеален моделируемый ею мир.

¹ От англ. *openspace* — большое офисное помещение, зал без перегородок с расположенными в нем рабочими местами.

² См. публикацию «Objects Have Failed» (2000 г.) и материалы конференции OOPSLA (Object-Oriented Programming, Systems, Languages and Applications) по данной теме в 2002 г.

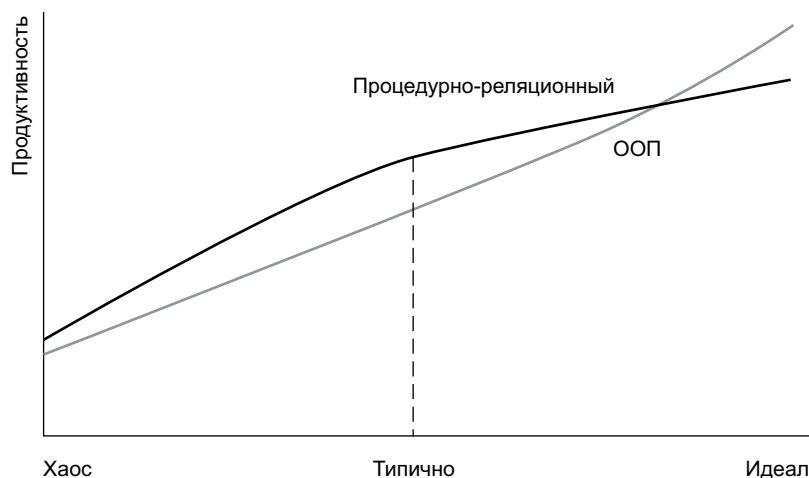


Рис. 3. Эмпирическое сравнение производительности процедурно-реляционного и объектно-ориентированного подходов в зависимости от достигнутой степени формализации моделируемого мира

Действительно, вспомним ещё раз Smalltalk. Его концепции выросли из задач построения графического интерфейса пользователя. Взглянув на любой оконный фреймворк, вы увидите искусственный мир, идеальный с точки зрения его авторов. Многоуровневые иерархии классов не воссозданы многолетним трудом классификации объектов окружающего мира, а выращены в виртуальных пробирках лабораторий разработчиков.

Учебники по ООП полны примеров, как легко и красиво решается задача отображения геометрических фигур на холсте с одним абстрактным предком и виртуальной функцией показа. Но стоит применить такой подход к объектам реального мира, как возникнет необходимость во множественном наследовании от сотни разношёрстных абстрактных заготовок. Объект «книга» в приложении для библиотеки должен обладать свойствами «абстрактного печатного издания», в магазине — «абстрактного товара», в музее — «абстрактного экспоната», в редакции, типографии, в службе доставки... Можете продолжить сами.

Попытки выпутаться из этой ситуации за счёт агрегации приводят к новым дивным мирам, существующим только в воображении разработчиков. Теперь объект «книга» это контейнер для чего-то продающегося, выдаваемого, хранящегося и пылящегося. Необходимо быстро менять контекст: в магазине вкладывать в книгу товар, в библиотеке — печатное издание, в отделе «книга-почтой» — ещё

какую-нибудь хреновину. Плодятся новые многоуровневые иерархии, но теперь уже не наследования (*is a*), а вложения (*is a part of*).

Изящнее выглядят интерфейсы. Но если в реальном мире книга, она и в музее — книга, то во вселенной интерфейсов «книга в музее» — неопознанный объект, пока не реализован соответствующий интерфейс «экспонат». Дальше интерфейсы пересекаются, обобщаются, и мы получаем ту же самую иерархию наследования, от которой сбежали. Но теперь это уже иерархия, во-первых, множественная, а во-вторых, состоящая из абстрактных классов без какой-либо реализации вообще (интерфейс, по сути, есть *pure abstract class*). Если же мы отказываемся от обобщения интерфейсов, то фактически оказываемся в рамках современных реализаций модульного программирования типа Оберон¹.

Тем не менее все три подхода применимы и могут дать хороший результат при высокой квалификации проектировщиков и наличии проработанных моделей предметной области.

Одна из причин подобных зловключений в том, что концепции, выдвигаемые ООП, на самом деле не являются его особенностями за исключением наследования реализации от обобщённых предков с виртуализацией их функций. И по несчастливому стечению обстоятельств именно наследование реализации является одним из основных механизмов порождения ада наследуемых ошибок, неявных зависимостей и хрупкого дизайна. Все остальные концепты от инкапсуляции и абстракции до полиморфизма имеются в вашем распоряжении без ООП. Полиморфизм с проекциями вместо таблиц наличествует даже в SQL.

Мой субъективный опыт подтверждает, что за исключением фреймворков весьма абстрактного уровня, сделанных «с чистого листа» небольшими группами профессионалов высокого класса, Объектно-Ориентированный Подход на практике в большинстве случаев превращает проект или продукт, переваливший за сотню-другую тысяч строк, в упомянутый Ад Паттернов, который, несмотря на формальную архитектурную правильность и её же функциональную бессмысленность, никто без помощи авторов развивать не может.

С другой стороны, любая неясность в постановке задачи вынуждает разработчиков сосредотачиваться не на её решении, а на архитектуре, позволяющей «без особых затруднений» менять логику приложения и переходить с расчёта зарплат колхоза на прогноз удоев фермы.

Результат неизменно стабильный...

¹ Оберон — семейство языков программирования высокого уровня, разработанных Никлаусом Виртом и его школой.

Особо хочу остановиться на тезисе уменьшения сложности при использовании ООП для создания фреймворков. Современное состояние дел — это платформа .NET с примерно *40 тысячами* классов и типов ещё в версии 3.5. Вдумайтесь, вам предлагают для выражения потребностей прикладного программирования язык с 40 тысячами слов, без учёта глаголов и прилагательных, называя такую технологию упрощением.

Александр Сергеевич Пушкин использовал в своем творчестве порядка 24 тысяч слов. Толковый словарь Ожегова содержит около 70 тысяч слов. Среднестатистический русский человек использует в повседневной жизни от 5 до 10 тысяч слов¹, из них только 3 тысячи являются общеупотребительными. Получается, что даже наследник гения Пушкина способен охватить менее половины предлагаемой технологии, при том что её словарь сравним с естественным языком!

Надо признать, входной порог использования ООП оказался гораздо выше, чем предполагалось в 1980–90-х гг. С учётом девальвации среднего уровня знаний «прогрессивных технологий», меняющихся по законам бизнеса квазимонополий, и прибывающих выпускников трёхмесячных курсов этот порог ещё и растёт с каждым годом.

Практика подтвердила, что технология, будучи обёрнутой в относительно безопасные языковые средства и жёсткие стандарты, допускает применение в больших проектах. С другой стороны, немалое число крупных проектов принципиально не используют ООП, например, ядро операционной системы Linux, Windows API или движок Zend уже упоминавшегося PHP. Язык C по-прежнему занимает первое место согласно статистике активности сообществ программистов², стабильно опережая второго лидера Java — например, индексы ноября 2012 показывают 19 % против 17 %.

Разумнее предположить, что реальную отдачу от ООП вы получите, только создав достаточно хорошие модели предметных областей. То есть тот самый минимально необходимый словарь и язык вашей системы для выражения её потребностей, доступный не только современному Пушкину от программирования. Модели будут настолько простыми и ясными, что их реализация не погрузит команду в inferнальные нагромождения шаблонных конструкций и непрерывный рефакторинг, а фреймворки будут легко использоваться при-

¹ Объем активного словаря образованного человека оценивается в среднем в 5–10 тысяч слов. «Сколько слов в русском языке?», Наука и жизнь, 2004, № 11.

² См. данные TIOBE Programming Community Index.

кладными программистами без помощи их авторов. Правда, тогда неизбежно встанет вопрос о необходимости использования ООП...

Осталось решить небольшую организационную задачу: где в некритичном проекте автоматизации «отдела X» найти бюджет на компетентного системного аналитика, технического архитектора и ведение хотя бы минимального набора проектной документации.

ORM, или объектно-реляционный проектор

Соккрытие базы данных, или Как скрестить ежа с ужом

Упомянув один из крупнейших столпов современного софтверостроения — мир ООП, нельзя обойти вниманием и другой — мир реляционных баз данных. Я намеренно вставил прилагательное «реляционные» применительно ко всем основным СУБД¹, хотя ещё в 1970-х годах такое обобщение было бы неправомерным.

Тем не менее именно реляционным СУБД удалось в 1980-х годах освободить программистов от знания ненужных деталей организации физического хранения данных, отгородиться от них структурами логического уровня и стандартизованным языком SQL² для доступа к информации. Также оказалось, что большинство форматов данных, которыми оперируют программы, хорошо ложатся на модель двумерных таблиц и связей между ними. Эти два фактора предопределили успех реляционных СУБД, а в качестве поощрительной премии сообщество получило строгую математическую теорию в основании технологии.

В отличие от реляционного мира, ООП развивалось инженерами-практиками достаточно стихийно, исходя из потребностей программистского сообщества, и потому никакой строгой теории под собой не имело. Имевшие место попытки подвести таковую под ООП задним числом терпели неудачу. Наилучшего результата добились авторы объявленного стандартом UML³, который, однако,

¹ Система Управления Базами Данных.

² Structured Query Language — язык структурированных запросов, также имеет название «сиквел», идущее от первой версии языка SEQUEL. Первый стандарт принят в 1986 г.

³ Unified Modeling Language — унифицированный язык моделирования, эклектичное объединение графических нотаций разных авторов, использовавшихся ими в софтверостроительных проектах на основе ООП.

в основном до сих пор используется в качестве иллюстрирующих код картинок. Но лучше плохой стандарт, чем никакой.

И реляционная и объектная модели относятся к логическому уровню¹ проектирования программной системы. Они ортогональны и по сути представляют собой два взгляда на одну и ту же сущность. Это значит, что вы можете реализовать одну и ту же систему, оставаясь в рамках только одного реляционно-процедурного подхода или же следуя исключительно ООП.

На практике сложилась ситуация, когда программы пишутся в основном с использованием ООП, тогда как данные хранятся в реляционных БД. Не касаясь пока вопроса целесообразности такого скрещивания «ежа с ужом», прием ситуацию как данность, из которой следует необходимость отображения (проецирования) объектов на реляционные структуры и обратно.

Ввиду упомянутого отсутствия под ООП формальной теоретической базы эта задача, в общем случае нерешаемая, выполнима в случаях частных. Компонент программной системы, реализующий отображение, называется *ORM*², или *объектно-реляционный проектор* — ОРП. Полноценный ORM может быть весьма нетривиальным компонентом, по сложности превосходящим остальную систему. Поэтому, хотя многие разработчики с успехом пользуются своими собственными частными реализациями, в отрасли за последние 10 лет появилось несколько широко используемых фреймворков, выполняющих в том числе и задачу проекции.

Обзор средств объектно-реляционной проекции выходит за рамки данной книги. Их и так достаточно в Сети, включая небольшой мой собственный [8], сделанный ещё в 2005 году, но не сильно устаревший. Поэтому последующие примеры будут в основном касаться фреймворка *NHibernate*.

В технологии отображения объектов на РСУБД есть очень важный момент, от понимания которого во многом зависит успех вашего проекта. Я не раз слышал мнение программистов, что для слоя домена³ генерируемый проектором SQL код является аналогом результата трансляции языка высокого уровня в ассемблер целевого процессора. Это мнение не просто глубоко ошибочно, оно быстрыми темпами ведёт команду к созданию трудносопровождаемых систем с врождёнными и практически неисправимыми проблемами производительности.

¹ В софтверостроении, как инженерной дисциплине, различают концептуальный, логический и детальный (физический) уровни проектирования.

² Object-Relational Mapping — объектно-реляционная проекция.

³ От англ. domain layer — слой объектов предметной области, часто называемых также бизнес-объектами.

Проще говоря, как только вы подумали о SQL как о некоем ассемблере по отношению к используемому языку ООП, вы сразу влипаете в очень нехорошую историю.

SQL — высокоуровневый декларативный специализированный язык четвертого поколения, в отличие от того же Java или C#, по-прежнему относящихся к третьему поколению языков императивных. Единственный оператор SQL на три десятка строк, выполняющий нечто посложнее выборки по ключу, потребует для достижения того же результата в разы, если не на порядок, больше строк на C#.

Такая ситуация приводит разработчиков ORM к необходимости создавать собственный SQL-подобный язык для манипуляции объектами и уже его транслировать в сиквел¹ (см. HQL²). Или использовать непосредственно SQL с динамическим преобразованием результата в коллекцию объектов.

В противном случае прикладной программист обречён на извлечение из БД и последующую обработку больших массивов данных непосредственно в своём приложении. Примерно так же обрабатывали табличные данные при отсутствии встроенного SQL разработчики на ранних версиях Clipper в конце 80-х годов. Там это называлось «навигационная обработка». Думаю, термин уместен и здесь.

В эпоху массового перехода с Clipper-подобных программ и файл-серверных технологий на клиент-серверные РСУБД многие приложения и их разработчики продолжали использовать навигационный подход. Приложения работали медленно, зачастую блокируя работу в многопользовательской среде. Потому что для эффективной работы с РСУБД нужно использовать подходы, ориентированные на обработку множеств на сервере, предполагающие наличие у разработчика умений работать с декларативными языками.

Тем не менее, получив в распоряжение ORM, программист зачастую возвращается к навигационным подходам обработки массивов данных вне РСУБД лишь с той разницей, что теперь этот массив, как хочется надеяться, является содержимым целой таблицы.

Почему? Недостаток знаний РСУБД пытаются возместить дополнительными уровнями абстракций. На деле же выходит обратное: уровни абстракции скрывают не детали слоя хранения объектов от программиста, а, наоборот, его некомпетентность в области баз данных от СУБД. До некоторого времени.

¹ Жаргонное название SQL-запроса. — *Примеч. ред.*

² Hibernate Query Language — SQL-подобный язык запросов, используемый в Hibernate/NHibernate.

Несмотря на толстый слой абстракций, предоставляемый ORM типа Hibernate, заставить приложение эффективно работать с РСУБД без знаний соответствующих принципов ортогонального мира и языка SQL практически невозможно.

Но попытки продолжаются. Одни по-прежнему разрабатывают проекторы для своих внутренних нужд, зачастую очень лёгкие. Другие ищут упрощение и выход в `poSQL`¹. Но в выигрыше пока остаются программисты и консультанты, обладающие «базоданными» компетенциями и зарабатывающие на тех, кто ими не обладает.

Как обычно используют ORM

На софтостроительных презентациях часто рисуют красивые схемы по разделению слоёв представления, бизнес-логики и хранимых данных. Голубая мечта начинающего программиста — использовать только одну среду и язык для разработки всех слоёв и забыть про необходимость знаний реляционных СУБД, сведя их назначение к некоей «интеллектуальной файловой системе». Аббревиатура SQL вызывает негативные ассоциации, связанные с чем-то древним, не говоря уже про триггеры или хранимые процедуры. На горизонте появляются добрые люди, с книгами признанных гуру о домен-ориентированной разработке² под мышкой, заявляющие новичкам примерно следующее: «Ребята, реляционные СУБД — пережиток затянувшейся эпохи 30-летней давности. Сейчас всё строится на ООП. И есть чудесная штука — ORM. Начните использовать её и забудьте про тяжёлое наследие прошлого!»

Ребята принимают предложение. Дальше эволюция разработки системы примерно следующая:

- Вначале происходит выбор ORM-фреймворка для отображения. Уже на этом этапе выясняется, что с теорией и стандартами дело обстоит плохо. Впору насторожиться бы, но презентация, показывающая, как за 10 минут создать основу приложения типа записной книжки контактов, очаровывает. Решено!

¹ `poSQL` — Not Only SQL, обобщённое название СУБД, не базирующихся на реляционной модели, интерес к которым возрос в последние годы прежде всего из-за сложности горизонтального масштабирования в традиционных РСУБД.

² Domain Driven Design — концепция разработки прикладных программ на основе домена — слоя объектов предметной области.

- Начинаем реализовывать модель предметной области. Добавляем классы, свойства, связи. Генерируем структуру базы данных или подключаемся к существующей. Строим интерфейс управления объектами типа CRUD¹. Все достаточно просто и на первый взгляд кажется вполне сравнимым с манипуляциями над DataSet² — тем, кто о них знает, конечно — ведь не все подозревают о существовании табличных форм жизни данных в приложении за пределами сеток отображения DBGrid.

Как только разработчики реализовали CRUD-логику, начинается основное действо. Использовать сиквел напрямую теперь затруднительно. Если не касаться стратегий отображения и проблем переносимости приложения между СУБД, по сути каждый SQL-запрос с соединениями, поднявшись в домен, сопровождается специфической проекцией табличного результата на созданный по этому случаю класс. По этой причине приходится использовать собственный язык запросов ORM. Нестандартный, без средств отладки и профилирования. Если он, язык, вообще имеется в данном ORM. Для поддерживающих соответствующую интеграцию среда .NET предоставляет возможность использовать LINQ³, позволяющий отловить некоторые ошибки на стадии компиляции.

Сравните выразительность языка на простом примере, который я оставляю без комментариев:

SQL

```
SELECT *
FROM task_queue
WHERE
    id_task IN (2, 3, 15)
    AND id_task_origin = 10
```

NHibernate HQL

```
IList<TaskQueue> queues = session
    .CreateQuery("from TaskQueue where Task.Id in (2, 3, 15) and TaskOrigin.Id = 10")
    .List<TaskQueue>();
```

¹ Create-Retrieve-Update-Delete — базовый набор манипуляций бизнес-объектами: «создать», «выбрать», «модифицировать», «удалить».

² DataSet — набор данных, в узком значении — двумерный динамический массив, заполняемый табличным результатом запроса к РСУБД.

³ LINQ — Language Integrated Query, технология Microsoft на основе SQL-подобного встроенного языка для манипуляции объектами в .NET-приложениях.

NHibernate без HQL с критериями

```
IList<TaskQueue> queues = session.CreateCriteria()  
    .Add(Expression.In("Task.Id", someTasks.ToArray()))  
    .Add(Expression.Eq("TaskOrigin.Id", 10))  
    .List<TaskQueue>();
```

LINQ (NHibernate)

```
IList<TaskQueue> queues = session  
    .Query<TaskQueue>()  
    .Where(q => someTasks.Contains(q.Task.Id) &&  
        q.TaskOrigin.Id == 10).ToList();
```

Внезапно оказывается, что собственный язык запросов генерирует далеко не самый оптимальный SQL. Когда БД относительно небольшая, сотня тысяч записей в наиболее длинных таблицах, а запросы не слишком сложны, то даже неоптимальный сиквел во многих случаях не вызовет явных проблем. Пользователь немного подождёт.

Однако запросы типа «выбрать сотрудников, зарплата которых в течение последнего года не превышала среднюю за предыдущий год» уже вызывают проблемы на уровне встроенного языка. Тогда разработчики идут единственно возможным путём: выбираем коллекцию объектов и в циклах фильтруем и об-считываем, вызывая методы связанных объектов. Или используем тот же LINQ над выбранным массивом. Количество промежуточных коротких SQL-запросов к СУБД при такой обработке коллекций может исчисляться десятками тысяч.

Триггер как идеальная концепция для NHibernate

Обычно разработчикам баз данных я рекомендую избегать необоснованного использования триггеров. Потому что их сложнее программировать и отлаживать. Оставаясь скрытыми в потоке управления, они напрямую влияют на производительность и могут давать неожиданные побочные эффекты. Пользуйтесь декларативной ссылочной целостностью (DRI¹) и хранимыми процедурами, пока возможно. А если ваш администратор баз данных склонен к параноидальным практикам «запрещено всё, что не разрешено», избегайте программировать на уровне СУБД, исключая критичные по производительности участки. Приходится говорить это с сожалением...

¹ От англ. Declarative Referential Integrity — декларативная ссылочная целостность, один из важнейших механизмов поддержания базы данных в непротиворечивом состоянии.

Однако стоит посмотреть на слой домена, живущего под управлением NHibernate, как становится ясно, что триггер в СУБД — это достаточно простая и хорошо документированная технология. В то время как NHibernate предлагает прикладному разработчику целый зоопарк триггероподобных решений.

Во-первых, имеется древний способ реализации классом домена интерфейса из пространства имён NHibernate.Classic. Например, `IValidate`. Вроде бы удобно: реализовал и делай проверки, генерируя исключения для отмены транзакции. Но вот незадача: при удалении объекта этот метод не срабатывает, нужно использовать другие подходы.

Во-вторых, после осознания авторами недостаточности `IValidate` и `ILifeCycle` была введена система прерываний (*interceptors*). Это уже больше, чем бесплатный хлеб на завтрак. Однако в обработчиках типа `Save` или `FlushDirty` в качестве аргументов используются массивы состояний объекта. То есть изменять сам объект в них напрямую нельзя: в общем случае это просто не срабатывает, но могут быть и побочные эффекты. Нужно, ни много ни мало, поискать индекс элемента в массиве имён свойств объекта, затем по найденному номеру изменить значение в другом массиве текущего состояния объекта. Что-то вроде такого кода:

Изменение свойств объекта в обработчике NHibernate

```
int index = Array.IndexOf(propertyNames, "PhoneNumber");
if (index != -1)
    state[index] = "(123)456789";
```

Создавать новые, извлекать, изменять или удалять существующие объекты с последующим сохранением внутри обработчика совершенно не рекомендуется. Кроме собственно гонок (*race condition*), когда обработчик одного класса создаёт другой, а тот что-то делает с первым, могут быть и другие эффекты, включая бесконечную рекурсию. Шаг вправо, шаг влево — стреляют боевыми и без предупреждения. Неплохая иллюстрация к теме декларируемой безопасности языка C# или Java. Рекомендуемая практика обхода ловушек такого рода — запрограммировать собственную защищённую (*thread safe*) очередь, куда складывать все созданные или изменённые объекты, а в событиях `BeforeCommit` или `AfterCommit` эту очередь обрабатывать.

В-третьих, механизм прерываний также признан несовершенным, после чего был введён механизм событий (*events*), коим, начиная со второй версии, всем следует пользоваться.

Дело в том, что в сессии вы можете зарегистрировать только один класс, реализующий обработчики прерываний. И если у вас достаточно много разной обработки, то получается так называемый «волшебный класс», который реализует всё. Это неудобно, даже если использовать класс в качестве пустого фасада.

Теперь же вы можете зарегистрировать неограниченное количество классов-слушателей (*listeners*), ожидающих то или иное событие и соответствующим образом реагирующих на него. Один класс может реализовывать несколько обработчиков событий, будучи зарегистрированным для прослушивания нескольких их типов.

С точки зрения архитектуры это несомненный плюс, реализацию можно разбить на независимые классы. Однако для снижения побочных эффектов, прежде всего рекурсии и гонок, не сделано ничего. В том же SQL Server, напомним, рекурсия в триггерах отключена по умолчанию, поскольку трудно сходу придумать случай, когда она нужна. А в событиях NHibernate каждый сам себе вредитель. При этом однозначной методики и документации нет, нескольким десяткам типов событий в официальной документации отведено меньше страницы. Существует огромное количество записей в блогах, тиражирующих одни и те же конкретные примеры — аудит, прежде всего. Но однозначной выверенной практики нет, в каждом конкретном случае надо проводить тесты. Например, для манипуляции объектами рекомендуется создавать дочернюю сессию, что также не всегда избавляет от побочных эффектов.

В итоге имеем плохо документированную нестабильную систему, которая при отладке в разы труднее столь нелюбимых разработчиками триггеров баз данных. Тем не менее, если разработчик пишет код слоя домена, альтернатив практически нет. Генерация скелета кода по модели облегчает работу, но риски возникновения ловушек многопоточной обработки по-прежнему остаются на совести рядового программиста. А создать такие ситуации несложно. Поэтому надо признать, что использование обработчиков событий слоя домена должно быть рекомендовано еще в меньшей степени, чем использование триггеров слоя хранения данных.

ORM на софтостроительной площадке

На короткое время судьба забросила меня в качестве консультанта в лоно одной софтостроительной фирмы, разработавшей и поддерживающей специализированную систему документооборота для управления жизненным циклом товаров. Система относительно небольшая по функционалу, а вот клиенты хоть и мало-

численные, но крупные, то есть способные упорно настаивать на своих требованиях, иногда противоречивых, аргументируя их соответствующим бюджетом.

В процессах взаимодействия фирм весьма отчётливо действуют физические законы всемирного тяготения. Небольшой планете-фирме, чтобы не упасть на большую, разбившись вдребезги, необходимо развить как минимум первую космическую скорость. В этом случае она будет стабильно вращаться вокруг большой в качестве спутника. Чтобы оторваться от поля тяготения большой планеты и начать самостоятельный полет требуется уже вторая космическая скорость.

В течение последних месяцев в фирме происходила попытка выйти на вторую космическую. Поскольку процесс, обеспечивающий первую космическую, был близок к тому, что называют экстремальным программированием, было принято решение продолжать в том же духе, назвав все это звонким словечком «скрам»¹.

В принципе, основные элементы процесса имелись в наличии. Коллективное владение кодом, также известное, как личная безответственность при его написании, утренние «пионерские линейки» вместо чётких спецификаций, практически полное отсутствие документации, частые, до одного раза в 1–2 недели, релизы и связанный с этим нескончаемый аврал, работа в тесном и жарком помещении общего зала. Последнее вызвано объективными причинами: для поддержания жизнедеятельности муравейника требуется всё больше работяг.

Вы резонно спросите: «А где рефакторинг?» Системе на тот момент исполнилось уже 2 года. Рефакторинг проводился раньше, но, в связи с тем, что его стоимость, прежде всего по требуемым срокам поставки, возрастала, количество реструктурируемого кода линейно уменьшалось. Это создало положительную обратную связь: реструктуризация становилась всё дороже.

На столах у некоторых программистов лежали книжки по рефакторингу от раскрученных апостолов веры в эволюционный дизайн. Но в связи с занятостью чётко выполнялась только первая часть моего любимого правила «Настоящий исследователь должен поменьше читать, чтобы иметь возможность больше думать головой». На вторую часть, к сожалению, у разработчиков не было времени.

Приведу пример подсистемы, для которой рефакторинг уже стал дороже полной переделки. Имелся модуль экспорта документов в форматы, пригодные для импорта системами клиентов уровня их внутренних АСУП². Коллективная

¹ От англ. SCRUM — одна из популярных методик так называемой «гибкой разработки».

² АСУП — Автоматизированная Система Управления Предприятием, англ. ERP — Enterprise Resources Planning.

ответственность привела к выбору написания императивного кода в размере 20 тысяч строк вместо разработки нескольких шаблонов XSLT¹ из нескольких сотен строк. Почему? Во-первых, опасались потери производительности, а во-вторых, не имели достаточной компетенции в XSL. Цикломатическая сложность² кода в отдельных методах превышала запредельное число 50 при рекомендованном пороге в 10–20. Глубина вложенности вызовов также была больше 10, при цикличности их части: **this** с верхнего уровня передаётся в качестве параметра, и где-то глубоко внизу дёргают этот **this** за какой-то метод. Объектно-ориентированная тарелка со спагетти.

О производительности следует сказать отдельно. Загрузка достаточно сложного документа перед его экспортом занимала порядка 30 секунд. Потому что было принято идеологическое решение «ни строчки SQL», несмотря на необходимость поддержки только одной РСУБД. Вся система работала через слой доступа³ под управлением NHibernate. Это был именно DAL, а не домен, так как парни не использовали всю мощь NHibernate, ограничиваясь отображением, и накручивали сверху слои бизнес-логики. При загрузке сложного документа с проверками подсистемы безопасности было насчитано порядка 20 тысяч (!) коротких SQL-запросов.

Почему именно такое решение? Было сказано некоторое количество красивых слов о чистоте объектной концепции. Это стандартный ответ. Тогда я просто предложил использовать в одном узком месте вместо сотен строк вложенных циклов сишарп-кода относительно короткий рекурсивный запрос из 40 строк сиквел-кода. Вид этого кода вначале вызвал у парней лёгкий ступор, а после совещания через сутки было принято решение отказаться от него. Но надо отдать должное: ребята честно признались, что после моего ухода никто не сможет этот сиквел-код поддерживать и модифицировать. Вот, собственно, и главная причина первоначального выбора. Красноречивое подтверждение тезиса о том, что слой объектной абстракции доступа к реляционной СУБД в большинстве случаев скрывает не базу данных от приложения, а некомпетентность разработчиков приложения в области баз данных.

Долго и неинтересно рассказывать, каким образом система с тремя сотнями таблиц умудрилась обрабаи миллионом строк C#-кода, для поддержки которого

¹ XSLT — англ. eXtensible Stylesheet Language Transformations — декларативный язык преобразования XML-документов. Стандартизован консорциумом W3C.

² Одна из метрик, обеспечивающая количественную оценку логической сложности программы.

³ DAL — Database Access Layer, слой абстракций для доступа к данным.

требуются всё новые разработчики. Клиенты проявляли недовольство, так как сроки срывались, а задержка медленно, но неуклонно росла. Два совещания, на которых генеральный директор, милейший мсьё, пытался реанимировать дух прежнего стартапа¹, искренне желая, чтобы все, от стажера до его ближайших замов, смело поднимали и доносили проблемы, по всей видимости, прошли впустую. Состояние постоянного стресса передалось даже мне, формально не несущему ответственности за результат. Но ведь ещё есть риски социального капитала, репутации, которая долго зарабатывается и очень быстро может обесцениться. Пришлось даже посидеть с мыслями об оптимальных путях выхода из миссии² за бокалом бургундского.

История закончилась типично. С большими усилиями систему дотянули до сдачи в эксплуатацию, хронические проблемы вследствие архитектурных просчётов названы особенностями, после чего развитие её было заморожено, а команда частично распущена, частично переориентирована на другие проекты.

Эмпирика

Возвращаясь к сюжету предыдущей главы, в качестве одной из метрик оценки качества реализации приложений корпоративной информационной системы можно принять соотношение числа таблиц в базе данных к числу тысяч строк кода программ без учёта тестов.

Чем меньше кода, тем лучше, это понятно. Например, качественная реализация слоя хранения использует DRI и прочую декларативность на уровне метаданных вместо императивного кодирования такой логики.

- Соотношение 1 к 1–2 примерно соответствует тому порогу, за которым начинается так называемый «плохой код».
- 1 к 3–4 — следует серьёзно заняться изучением вопроса переделки частей системы.
- 1 к 5 и более — надеемся, что случай нестандартный (сложные алгоритмы, распределенные вычисления, базовые подсистемы и компоненты реализуются самим разработчиком), либо «врач сказал — в морг».

¹ От англ. startup — новообразованное инновационное малое предприятие, термин чаще всего употребляется в наукоемких областях деятельности.

² Миссия — временная работа консультанта на площадке заказчика.

ВЦКП в облаках

Можете ли вы представить себе личный автомобиль, передвигающийся во время поездок с предусмотренной скоростью лишь 1 час из 10? Именно таков ваш персональный компьютер, планшет или смартфон. Его вычислительная мощность используется в среднем на 5–10 % даже на рабочем месте в офисе. Более пропорционально расходуется оперативная память. Операционная система Windows NT 4 свободно работала на устройстве с ОЗУ¹ объёмом 16 Мбайт. Windows 7 требуется уже минимум 1 Гбайт. Правда, я не уверен, что Windows 7 хотя бы по одному параметру в 60 раз лучше предшественницы.

Широко известный в узких кругах своими несколько провокационными книгами² публицист Николас Карр пишет: «Отделы ИТ в компаниях уходят в прошлое, а сотрудники соответствующих специальностей останутся не у дел из-за перехода их работодателей на сервис, предоставляемый по принципу коммунальных услуг»[4].

В середине 1990-х годов выходила аналогичная по своему пропагандистскому назначению книга «Стратегии клиент-сервер»[5], где описывались совершенно противоположные тенденции перехода от централизованных систем к распределенным ввиду избытка подешевевших мощностей на терминалах.

С развитием каналов связи вектор децентрализации вычислительных ресурсов вновь сменился на обратный. Но пользователи уже получили в своё распоряжение многочисленные устройства, не использующие и десятой части своих возможностей. Чтобы предполагать, как изменится направление в будущем, следует вспомнить собственную историю.

На заре массового внедрения АСУ на предприятиях в 1970-х годах советские управленцы и технические специалисты предвидели многое. А именно, сосредоточение ресурсов в вычислительных центрах, где конечные пользователи будут получать обслуживание по решению своих задач и доступ. Доступ по простым терминалам, в ту эпоху ещё алфавитно-цифровым. Такова была концепция ВЦКП — Вычислительного Центра Коллективного Пользования.

В 1972 году впервые прозвучали слова о государственной сети вычислительных центров, объединяющих региональные ВЦКП. Для чего, по большому

¹ Оперативное Запоминающее Устройство, англ. RAM — Random Access Memory.

² Например, «Does IT Matter? Information Technology and the Corrosion of Competitive Advantage» (Есть ли толк от ИТ? Информационные технологии и выветривание конкурентных преимуществ), издательство Harvard Business Review, 2004 г.

счёту, и потребовалась унифицированная техническая база в виде ЭВМ ЕС¹ и позднее СМ ЭВМ². Разумеется, подобные проекты существовали и в США, иначе откуда бы взяться IBM System/360. Но, в отличие от советской программы, американцы на той стадии развития рынка и технологий потенциально могли охватить только околোগосударственные и государственные структуры. В СССР же, напомним, все предприятия и организации были государственными за редкими формальными исключениями. Охват экономики планировался куда более полный.

Чем занимались тогда проектировщики, техники и организаторы? Примерно тем же самым, чем занимаются сейчас продвигающие на рынок системы «облачных» (*cloud*) вычислений, за исключением затрат на рекламу. Создавали промышленные вычислительные системы, отраслевые стандарты и их реализации. Чтобы предприятия-пользователи, те, кому это экономически нецелесообразно, не содержали свои ВЦ, а пользовались коллективными, ведь пироги должен печь пирожник, а не сапожник. Чтобы легче было собирать статистическую информацию. Чтобы снижать издержки на инфраструктуру, оборудование и эксплуатацию.

Если частично заменить «большие ЭВМ» на «кластеры из серверов», добавить взращенную за последние 10 лет широкую полосу пропускания общедоступных сетей на «последнем километре»³, а в качестве терминала использовать веб-браузер или специальное приложение, работающее на любом персональном вычислительном устройстве, от настольного компьютера до коммуникатора, то суть не изменится. Облако — оно же ВЦКП для корпоративного и даже массового рынка.

В чем состояли просчёты советской программы ВЦКП? Их два. Один крупный: проглядели стремительную миниатюризацию и, как следствие, появившееся обилие терминалов. Хотя отдельные центры, как, например, петербургский «ВЦКП Жилищного Хозяйства», основанный в 1980 году, действуют до сих пор, мигрировав в 1990-х от мейнфреймов к сетям ПК. Второй: просчитались по мелочам, не спрогнозировав развал страны с последующим переходом к состоянию технологической зависимости.

Авторам мемуаров о создании в СССР первых ВЦКП [7], возможно, будет не только досадно, но и приятно. Досадно за опередившие время своего техно-

¹ ЭВМ Единой Серии, относились к классу больших универсальных ЭВМ.

² Система Малых ЭВМ.

³ Английский термин *last mile* или *last kilometer* — канал, соединяющий конечное оборудование клиента с узлом доступа оператора связи.

логического воплощения концепции. Приятно за реализацию идеи — лучше поздно, чем никогда.

Впрочем, для истории это уже неважно. Нам интереснее попытаться усмотреть тенденции технологического маятника, резко качнувшегося в 1980–90-х годах в сторону децентрализации и теперь возвращающегося на позиции годов 1970-х в качественно новой ситуации обилия дешёвых терминалов с их избыточными мощностями и общедоступными высокоскоростными каналами связи.

Программировать распределённое приложение сложнее, чем централизованное. Не только технически, но и организационно. Но в качестве выигрыша получаем автономию сотрудников, рабочего места и отсутствие необходимости поддержки службы в состоянии 24×7.

Наиболее очевидное применение децентрализации — автономные программы аналитической обработки данных. Мощность терминала позволяет хранить и обрабатывать локальную копию части общей базы данных, используя собственные ресурсы и не заставляя центральный сервер накаляться от множества параллельных тяжёлых запросов к СУБД. Пример из практики мы рассмотрим в следующих главах.

SaaS и манипуляции терминами

Софтостроение — производственный процесс с высокой долей НИОКР¹. Наукоёмкий процесс, или, как его ещё называют, высокотехнологичный. На выходе — продукт: программа, программный пакет, система или комплекс.

По причине развития общедоступных каналов связи работать с программами можно с удалённого терминала, в роли которого выступает множество устройств: от персонального компьютера до мобильного телефона. Эксплуатацию же программ ведут поставщики услуг в «облаках» — современные ВЦКП. Это и есть «программа как услуга», SaaS².

Значит ли это, что софтостроительные фирмы теперь, как утверждают некоторые маркетологи, «производят услуги»? Разумеется, нет.

Во-первых, услуги не производят, их оказывают. Во-вторых, услуга от продукта, материального товара, принципиально отличается минимум по трём пунктам:

¹ НИОКР — Научно-Исследовательские и Опытно-Конструкторские Работы.

² От англ. software as a service.

- услуги физически неосязаемы;
- услуги не поддаются хранению;
- оказание и потребление услуг, как правило, совпадают по времени и месту.

В чем же фокус? Фокус в том, что классическая цепочка «производитель — конечный потребитель» подразумевает, что потребитель сам приобретает нужную программу и право на использование, сам её устанавливает и эксплуатирует. Разумеется, между производителем и потребителем может быть много перепродавцов и посредников, а вокруг — консультантов, но ситуация от этого принципиально не меняется.

В схеме «программа как услуга» посредник (ВЦКП) берет на себя эксплуатацию программного продукта, то есть его установку, настройку, содержание, обновление и т. п., предлагая конечному пользователю услугу по доступу к собственно функциональности этой программы.

Это значит всего лишь, что софтверисты продолжают производить продукт. Но конечным пользователем для них является ВЦКП в «облаках», оказывающий на базе этого продукта услуги своим клиентам. Программа — это продукт, как и многие другие, например автомобиль, позволяющий на своей основе оказывать такие полезные услуги, как прокат, извоз или транспортировка.

От CORBA к SOA

Признаю, что название главы логически не совсем корректно, поскольку CORBA¹ — одна из технологий создания сервис-ориентированной архитектуры, SOA², оно отражает хронологию развития событий. Поскольку маркетинговый шум вокруг SOA стих, жужжащие словечки вроде *«loose coupling»* подзабылись, а в последние годы стало модным даже говорить о том, что «COA выдохлась», можно вернуться к сюжету в спокойной обстановке на уровне собственно технологии. Мне слово «COA» напоминает другое из 1980-х годов, СОИ — Стратегическая Оборонная Инициатива, оказавшаяся пропагандистским пузырём

¹ От англ. Common Object Request Broker Architecture — технологический стандарт разработки распределённых приложений, продвигаемый консорциумом OMG.

² От англ. SOA — Service Oriented Architecture.

игры в «звёздные войны» администрации американского президента Рейгана. Просто ассоциации, ничего личного.

Предшественником современной СОА на базе веб-служб, с полным правом можно считать CORBA. Это сейчас задним числом обобщают подходы, представляя СОА как набор слабосвязанных сетевых служб, реализовать которые можно по-разному. А тогда, в середине 1990-х, вокруг CORBA стоял достаточно сильный шум продавцов волшебных палочек, но никаких упоминаний СОА ещё не было. Только начинал своё бурное развитие интерактивный веб динамического контента, приём заказов через интернет-магазин считался «крутой фишкой» для компании, а где-то в недрах Microsoft из XML-RPC¹ тихо прорастал SOAP².

CORBA имела очевидные достоинства, прежде всего это *интероперабельность*³ и отраслевая стандартизация не только протоколов (шины), но и самих служб и общих механизмов — *facilities*. Например, служб имён, транзакций, безопасности, хранения объектов, конкурентного доступа, времени и т. п. Однажды реализованный программистами сервис мог использоваться многими системами без какой-либо дополнительной адаптации и ограничений с их стороны. Следует отметить, что фактически проигнорированная отраслью CORBA 1.0 не была интероперабельной и предназначалась только для языка С.

Почему же и CORBA 2, продержавшись несколько лет в основном потоке технологий, была оттеснена на обочину? Причин много, весьма подробный анализ описан в статье одного из разработчиков стандарта [14]. Мне же, как участнику разработки реальной системы на основе технологии, хотелось бы выделить следующие:

- Стандарт поддерживался многими корпорациями, поэтому развитие требовало длительного согласования интересов всех участников. Некоторые из них продвигали свои альтернативы, например Sun Java RMI и Jini, Microsoft DCOM.

¹ От англ. Extensible Markup Language Remote Procedure Call — XML-вызов удалённых процедур. Стандарт и протокол вызова удалённых процедур, использующий XML для сообщений и HTTP в качестве транспорта.

² От англ. Simple Object Access Protocol — протокол обмена сообщениями и вызова удалённых процедур на базе, прежде всего, HTTP и XML. Используется для разработки веб-служб.

³ От англ. interoperability — способность различных по архитектуре, аппаратным платформам и средам исполнения систем к взаимодействию.

- Несмотря на реальную поддержку интероперабельности, множества языков и сред программирования, основным средством разработки оставался C++. Но код на C++, манипулирующий инфраструктурой CORBA и службами, является излишне сложным по сравнению с той же Java или даже Delphi. Практиковавшие подтвердят, остальным будет достаточно взглянуть на примеры в Сети. А Java-программисты не спешили использовать CORBA из-за упомянутых альтернатив.
- Запоздалая (1999 год), объёмная и весьма сложная спецификация компонентной модели. Java-сообщество к тому времени обладало альтернативой в виде EJB¹ с открытыми и коммерческими реализациями.

Кто знает, откажись тогда корпорация Sun от роли единственно правильного хранителя своей технологии, сосредоточься она на интеграции с CORBA и реализации спецификаций, возможно, её история не закончилась бы через 10 лет поглощением со стороны Oracle. Но Sun тогда, на пике бума доткомов², предпочла строить свой собственный мир, планируя накрыть им всю отрасль. Насколько простой оказалась придуманная в корпорации реальность, можно судить по фрагменту из статьи 2002 года «Страдает ли Java от сложности?» [16].

Новоиспечённый инженер после пяти лет обучения, имеющий базовые знания Java, XML и UML, после своего прихода на предприятие и перед началом работы был привлечён в понедельник к чтению публичного отчёта по J2EE v1.4 на 228 страницах.

На шести первых страницах этого отчёта значились ссылки на EJB, JSP, JMS, JMX, JCA, JAAS, JAXP, JDBC, JNDI. Эта новая версия J2EE и веб-служб предполагала знание концепций SOAP, SAAJ, JAX-RPC и JAXR. Каждый из этих акронимов имеет соответствующую спецификацию.

Спецификация EJB 2.1 — это документ PDF на 640 страницах, который будет прочтён во вторник. Среда будет посвящена чтению документации по сервлетам Servlet 2.4 на 307 страницах. В четверг он штурмует документ по JSP 2.0 на 374 страницах. И так далее...

¹ Enterprise Java Beans — спецификация технологии написания и поддержки серверных компонентов в Java.

² От англ. dot-com, во время роста количества интернет-компаний в конце 1990-х годов, многие из них регистрировались в домене .com, отсюда и название.

После месяца интенсивных чтений наш инженер наконец готов начать продуктивную работу...

Я добавил бы, что указанные в статье оценки как временных рамок, так и продуктивности работы излишне оптимистичны. Но вернёмся к CORBA.

Из небольшого списка известных в тот период открытых реализаций (Orbus, MICO, Robin) нами в продукте использовался omniORB. В комплекте к третьей версии выпуска 2000 года шла единственная реализованная служба именования (COS naming services) и библиотека синхронизации потоков — минимум, чтобы просто запустить разработку «с нуля». Впрочем, небольшой команде стартапа этого вполне хватило.

В начале 2000-х в ИТ-отрасли разразился кризис лопнувшего пузыря интернет-компаний, продвигать сложные и дорогие инфраструктуры, а коммерческие реализации CORBA стоили порядка тысячи долларов за рабочее место, стало очень трудно. Постепенно из кустов начали выкатывать на сцену роуаль веб-сервисов, который из-за проблем с CORBA должен был стать новой платформой интеграции служб и приложений в гетерогенной среде. Концепции быстро придумали многообещающее название COA. Надо сказать, что роуаль не стоял без дела и в кустах: разработка веб-служб и развитие XML велись в конце 1990-х и активно продолжились в 2000-х годах. Поскольку это была технология, хотя и весьма базового уровня, но относительно простая, дешёвая и открытая не только на уровне спецификаций, но и в виде множества реализаций.

Говоря о базовом уровне технологии, я имею в виду не столько отсутствие сравнимой с CORBA функциональности и номенклатуры служб и средств¹, сколько необходимость программистам самим надстраивать над этим базисом многое из того, что было даже в самом минимальном варианте CORBA.

Веб, точнее, его основа, HTTP — среда без состояния и пользовательских сессий. В общедоступном Интернете такое решение было вызвано соображениями нагрузки, поскольку максимальное число запросов к серверу теоретически равно количеству всех устройств в сети. В корпоративной же системе нагрузку на службу можно (и нужно) рассчитать гораздо точнее. В итоге программистам, не связанным с веб-разработкой для Интернета, приходится восполнять недостаток средств протокола надстройками поверх него костылей, например, постоянно гоня контекст и состояние в сообщениях или симулируя сессии по тайм-ауту.

¹ См., например, список стандартов OASIS для веб-служб.

В CORBA сессии поддерживались средой без дополнительных усилий. Если в частных случаях возникали вопросы нагрузки на поддержку соединений при большом количестве клиентов, они решались так же просто, как и в среде СУБД: приложение самостоятельно отсоединялось от сервера, выполнив пакет необходимых запросов. При желании нетрудно было также организовать и принудительное отсоединение по истечении заданного периода пассивности. Но для корпоративной службы, напомним, речь идёт обычно о десятках и сотнях активных сессий, поддержка которых в большинстве случаев укладывается в ресурсы серверов.

Неприятным следствием отсутствия сессий стала невозможность поддержки транзакций при работе с веб-службами. Для решения проблемы в рамках ООП необходим шаблон «Единица работы» (*unit of work*), суть которого в передаче веб-службе сразу всего упорядоченного множества объектов, подлежащих сохранению в управляемой сервером транзакции.

В общем случае шаблон является аналогом пакетной обработки транзакций, необходимой для сокращения времени жизни единичной транзакции. Например, в репликации данных между СУБД сиквел-операции передаются пачками. Но если раньше такой подход был разновидностью оптимизации и средством избавления от толстых транзакций, то теперь его необходимо было использовать всегда, вместо любой транзакции вообще.

Давайте сравним близкий к реальному псевдокод на сторонах клиента в рамках CORBA с псевдокодом в среде веб-служб. При поддержке сессии все достаточно прозрачно и не нуждается в комментариях.

Псевдокод транзакции в среде CORBA

```
CosTransactions.Current current = CosTransactions.CurrentHelper.Narrow(
    orb.ResolveInitialReferences("TransactionCurrent"));
current.Begin();
try
{
    store1.Remove(product, quantity);
    store2.Append(product, quantity);
    current.Commit();
}
catch (Exception e)
{
    current.Rollback();
    ShowError("Ошибка выполнения операции: " + e.toString());
}
```

В среде веб-служб в программе-клиенте приходится надстраивать абстракции DTO¹. А в серверном приложении, где используются соединения, например, с СУБД или монитором транзакций, необходимо фактически дублировать предыдущий код с раскруткой объекта — единицы работы (unit of work) в реальную транзакцию.

Псевдокод транзакции в среде веб-служб

```
StoreServiceClient storeServiceClient = new StoreServiceClient(url);
StoreOperationDTO operation1 = storeServiceClient.CreateOperation(store1.Id);
operation1.Type = StoreOperations.Remove;
operation1.ProductId = product.Id;
operation1.Quantity = quantity;
StoreOperationDTO operation2 = storeServiceClient.CreateOperation(store2.Id);
operation2.Type = StoreOperations.Append;
operation2.ProductId = product.Id;
operation2.Quantity = quantity;

UnitOfWork uow = new UnitOfWork();
uof.RegisterDirty(operation1);
uof.RegisterDirty(operation2);
try
{
    storeServiceClient.ProcessOperations(uow);
}
catch (Exception e)
{
    ShowError("Ошибка выполнения операции: " + e.ToString());
}
```

Вторым «упрощением» стал переход от понятных прикладному программисту деклараций интерфейсов объектов и служб на языке IDL² к WSDL³ — описаниям, ориентированным, прежде всего, на обработку компьютером. Сравним декларации складской службы, возвращающей по запросу текущее количество товарных позиций.

Декларация службы в CORBA IDL

```
module StockServices
{
    typedef float CurrentQuantity;
```

¹ От англ. Data Transfer Object — простой объект без методов (по сути — структура), служащий для передачи состояния между клиентом и сервером.

² От англ. Interface Definition Language — язык спецификаций интерфейсов, синтаксически близкий к декларациям на C++.

³ От англ. Web Services Description Language — язык описания веб-служб и доступа к ним, основанный на XML.


```

struct QuantityRequest
{
    string stockSymbol;
};

interface StockInventoryService
{
    CurrentQuantity getCurrent(in QuantityRequest request);
};
};

```

Декларация службы в WSDL

```

<?xml version="1.0" encoding="utf-8"?>
<definitions name="StockInventoryService"
    xmlns:sqs="http://mycompany.com/stockinventoryservice.wsdl"
    xmlns:sqsxsd="http://mycompany.com/stockinventoryservice.xsd"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
  <wsdl:types>
    <xsd:element name="CurrentQuantity">
      <xsd:complexType>
        <xsd:all>
          <xsd:element name="stockSymbol" type="string"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="CurrentQuantity">
      <xsd:complexType>
        <xsd:all>
          <xsd:element name="quantity" type="float"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>
  </wsdl:types>

  <wsdl:message name="getCurrentInput">
    <wsdl:part name="body" element="sqsxsd:CurrentQuantity"/>
  </wsdl:message>
  <wsdl:message name="getCurrentOutput">
    <wsdl:part name="body" element="sqsxsd:CurrentQuantity"/>
  </wsdl:message>

  <wsdl:portType name="StockInventoryServicePortType">
    <wsdl:operation name="getCurrent">
      <wsdl:input message="sqs:getCurrentInput"/>
      <wsdl:output message="sqs:getCurrentOutput"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="StockInventoryServiceSoapBinding"
    type="sqs:StockInventoryServicePortType">

```

продолжение ⇨

```

<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="getCurrent">
  <soap:operation soapAction="http://mycompany.com/getCurrent"/>
  <wsdl:input>
    <soap:body use="literal"/>
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal"/>
  </wsdl:output>
</wsdl:operation>
</wsdl:binding>

<wsdl:service name="StockInventoryService">
  <wsdl:port name="StockInventoryServicePort"
    binding="sqs:StockInventoryServiceBinding">
    <soap:address location="http://mycompany.com/StockInventoryService"/>
  </wsdl:port>
</wsdl:service>
</definitions>

```

Третьим «усовершенствованием» стал отказ от автоматической подгрузки связанных объектов¹ в пользу исключительно ручного управления процессом.

В CORBA объекты функционируют на сервере, тогда как на клиенте находится только соответствующая заглушка (*stub*). То есть вы в программе вызываете какой-то метод, а на самом деле происходит обращение к серверу, вызов соответствующего метода у серверного объекта и возврат результата на клиента с возможным обновлением состояния локальных полей заглушки. Аналогично со свойствами объектного типа: связанный объект подгружается по мере необходимости. Всё это происходит прозрачно для программиста, которому не нужно вмешиваться в процесс взаимодействия, но желательно знать, что стоит за манипуляциями с заглушкой на клиенте. Соответственно, существует соблазн вместо реализации на сервере новой функции службы написать код непосредственно на клиенте, благо сделать это легко. Тогда, например, обработка достаточно большой коллекции объектов в цикле может вызвать интенсивный обмен сообщениями с сервером и возникновение узкого места в системе. Аналогичная проблема плохой реализации имеется и при работе приложения напрямую с СУБД.

В среде веб-сервисов вопрос с «нерадивым программистом» решили радикально — отменой самой возможности написать такой код. Несмотря на то что в 80 % случаев имевшаяся автоматическая загрузка была уместной и здорово сокращала программу.

¹ Также называется «ленивой инициализацией», от англ. lazy loading.

Как уберечь кукурузу от насекомых-вредителей? Очень просто: выкосить её всю, к чертям. Вредители придут, а кушать нечего.

Возвращаемые веб-службами объекты не связаны с серверными за их отсутствием. Потому что у серверной части приложения нет состояния и, соответственно, не может быть никаких объектов в принципе. Обмен сообщениями происходит как и в обычной веб-среде: запрос—ответ без поддержки сессии. Общеупотребительная практика — использование DTO для передачи состояния объектов от клиента к серверу и обратно. Но DTO не содержит никаких ссылок на другие объекты, кроме вложенных. Его структура состоит из полей скалярных типов, разрешённых стандартом, вложенных объектов и массивов. Соответственно, вы не можете прозрачным образом динамически подгрузить недостающий объект, для чего придётся явным образом вызывать службу.

Прозрачная загрузка объектов в клиентском CORBA-приложении

```
BookGroup group = catalog.getBookCategory("Программирование");
Book[] books = group.getItems(); // один вызов сервера
foreach(Book book in books)
{
    ShowInfo(book.Name + ": ");
    ShowInfo(book.getPopularity().getVotesCount()); // два вызова
}
```

Работа клиентского приложения с DTO в среде веб-служб

```
BookGroupServiceClient groupClient = new BookGroupServiceClient(url1);
BookGroupDTO group = groupClient.GetBookCategory("Программирование");
BookServiceClient bookClient = new BookServiceClient(url2);
BookDTO[] books = bookClient.GetByGroupId(group.Id);
foreach(BookDTO book in books)
{
    PopularityServiceClient popularityClient = new PopularityServiceClient(url3);
    PopularityDTO popularity = popularityClient.GetByBookId(book.Id);
    int votesCount = popularityClient.GetVotesCount(popularity.Id);
    ShowInfo(book.Name + ": ");
    ShowInfo(votesCount);
}
```

Сила CORBA проявляется в том, что технология может работать и как в приведённых примерах, то есть с реализацией элементов полноценного многопоточного сервера приложений, и аналогично веб-службам, обрабатывая в сервисах объявленные в интерфейсах структуры, напоминающие DTO.

Вне контекста «автоматизированного бардака»¹ современные заявления о том, что СОА не оправдала возложенных на неё надежд, свидетельствуют о том, что и выбранная для неё модель веб-служб не стала решением проблем взаимодействия приложений в корпоративной среде. Ожидает ли нас новое пришествие CORBA в виде облегчённой её версии — покажет время. Поищите в Интернете по ключевым словам Web ORB — обнаружите немало интересного.

Прогресс неотвратим

Войны не будет, но будет такая борьба за мир,
что камня на камне не останется!

Из анекдота времён холодной войны

Вы думаете, что большие ЭВМ вымерли или вымирают? Попытаюсь вас если не разубедить, то хотя бы проинформировать.

В 2011 году 93 % респондентов крупных компаний отметили, что интенсивность использования мейнфреймов в их деятельности увеличивается или, как минимум, стабильна, и они остаются критически важной платформой для ЦОДов и «облачных» систем. 62 % (в 2010-м таковых было 56 %) отмечают общий рост данного рынка, 47% полагают, что его развитие хорошо стимулируется новыми задачами и новым программным обеспечением².

Две трети опрошенных отводят мейнфреймам стратегическую роль, которая только увеличивается; 42 % отмечают особую важность мейнфреймов для облачных приложений, а 65 % даже намерены внедрять в управление этими машинами мобильный подход³.

¹ Одно из основных правил автоматизации предприятий, приписываемое основоположнику кибернетики в СССР В. М. Глушкову, звучит как «Беспорядок автоматизировать нельзя!»

² Данные опроса BMC Software, 2012 г.

³ Отчёт CA Technologies «The Mainframe as a Mainstay of the Enterprise», 2012 г.

Согласно данным IDC по региону EMEA, в классе решений high-end (системы ценой \$250 тыс. и выше) мейнфреймы опережают другие платформы pop-x86, имея по результатам 2011 года долю рынка 45 % в стоимостном выражении. При этом в данном регионе и в данном классе серверов средняя цена RISC-сервера — 580 тыс. долл., EPIC-сервера — 830 тыс. долл., мейнфрейма — 2,1 млн долл. Чем выше требования к производительности и защищенности системы (и цена, которую согласны платить за это заказчики), тем чаще выбор делается в пользу мейнфрейма... Если взять одну из наиболее мощных экономик Западной Европы — Германию, то там в 2011-м на рынке систем pop-x86 класса high-end доля мейнфреймов в количестве поставленных систем составляла 33 %, а в денежном выражении — 66 %.

Такие цифры должны впечатлять, особенно на фоне демонтированных в 1990-х годах на драгметаллы советских ЭВМ. Безукоризненно проведенная операция по зачистке национального рынка с последующим его заполнением экспортной продукцией самого низкого ценового сегмента — серверами и ПК на базе x86.

Наиболее ходовым термином в софтостроении является «новые технологии». По умолчанию новые технологии олицетворяют прогресс. Но всегда ли это так? Даже если не всегда, то в условиях квазимонополий, поделивших рынки корпораций, отказаться от «новых» технологий рядовым разработчикам непросто, особенно работающим в сфере обслуживания под руководством менеджеров среднего звена с далёким от технического образованием, оперирующих понятиями освоения и расширения бюджета и массовости рынка специалистов, а не технологической эффективностью.

Произошла тихая революция. Ещё 15–20 лет назад сессии крупных поставщиков на конференциях разработчиков были своеобразным мастер-классом, где на бета-стадии испытывалась реакция аудитории на предлагаемые изменения. Сегодня повестка дня состоит в постановке перед фактом новой версии платформы, показе новых «фишек» и оглашении списка технологий, которые больше не будут развиваться, а то и поддерживаться. Действительно, солдаты от софтостроения не должны рассуждать. Они несут службу и должны молча овладевать оружием, закупкой которого занимаются генералы в непрозрачном договоре с поставщиками. Экономика потребления обязана крутиться, даже если в ней перемальваются миллиардные бюджеты бесполезных трат на модернизацию, переделку и переобучение.

.NET

Цифры версий и релизов фреймворка .NET меняются со скоростью, заметно превышающей сроки отдачи от освоения и внедрения технологий. В качестве положительного момента отмечу тот факт, что можно перескочить со второй версии сразу на четвёртую, минуя третью и третью с половиной. Правда, уже анонсирована пятая.

Давайте подумаем, кто же выигрывает в этой гонке кроме самой корпорации, пользующейся своим квазимонопольным положением:

- **Услуги по сертификации.** Прямая выгода.
- **Консультанты и преподаватели курсов.** Сочетание прямой выгоды с некоторыми убытками за счёт переобучения и очередной сертификации.
- **Программисты в целом.** Состояние неопределённости. Выбор стоит между «изучать новые возможности» и «решать задачи заказчиков». А если изучать, то как не ошибиться с перспективой оказаться у разбитого корыта через пару лет.
- **Разработчики в заказных проектах.** Прямые убытки. За пару лет получен опыт работы с технологиями, признанными в новом фреймворке наследуемыми, то есть не подлежащими развитию, хорошо, если поддерживаемыми на уровне исправления критичных ошибок. А ведь всего 2–3 года назад поставщик убеждал, что эти технологии являются перспективными, важными, стратегическими и т. п. Необходимы новые инвестиции в обучение персонала и преодоление появившихся рисков.
- **Разработчики продуктов.** Косвенные убытки. В предлагаемом рынку продукте важна функциональность и последующая стоимость владения. На чём он написан — личное дело компании-разработчика. Тем не менее заброшенную поставщиком технологию придётся развивать за свой счёт или мигрировать на новую. Скорее второе: в 2012 году по прежнему работает приложение 15-летней давности, использующее DDE¹, тогда как совместимость OLE Automation между версиями Office не гарантирована.

¹ Dynamic Data Exchange — механизм взаимодействия приложений в операционных системах Windows и OS/2. Хотя этот механизм до сих пор работает в последних версиях Windows, он давно заменён на OLE, COM и Microsoft OLE Automation.

Риторически, подобно герою кинокомедии, можно вопрошать: «Минуточку, за чей счёт этот банкет?» И ответ будет аналогичный оригинальному, подвергнутому цензуре в фильме.

Софтостроителю должно быть понятно, что менять технологии и концепции рискованно, потому что новые ещё сырые, а брошенные или отодвинутые на второй план «старые» так и не успели достичь зрелости. 2–3 года — минимальный срок для появления первых промышленно работающих систем и, соответственно, специалистов по их разработке. А не специалистов по чтению обновлённой версии MSDN и книжек по учебным курсам. Поэтому нормальный цикл концептуальных изменений 5–7, а то и 10 лет.

Вынужденный совет в такой ситуации дают авторы книги «Прагматичный программист» (Pragmatic Programmer), в буквальном переводе «Не кладите все свои технические яйца в одну корзину»¹.

Office 2007

Как известно, Microsoft изменила интерфейс в Office 2007. Вместо привычных меню появились многочисленные закладки лент панелей инструментов с крупными пиктограммами. По словам Microsoft, это сделано для облегчения работы начинающим пользователям.

Хорошо, возможно, начинающим жить в офисном пакете это полезно или безразлично. А мы, давние пользователи, заканчиваем в нём жить, что ли? Кроме проблем с интерфейсом возникли проблемы открытия файлов в Office 2003. Повторилась ситуация с версией Office 97, когда Microsoft пришлось в срочном порядке выпускать конвертер для Office 95, позволяющий открывать в нём файлы новых форматов.

Разве трудно было предусмотреть возможность выбора между старым и новым интерфейсом, как это было сделано в Windows XP и 7? Зато всего за 30 долларов вам предложат купить программку третьей фирмы Classic Menu, которая возвратит старый интерфейс. Остап Бендер со своими относительно честными способами отъёма денег у населения мог бы гордиться последователями.

Крупные корпорации не спешили с обновлениями, соблюдая внутренние стандарты. В рамках одного проекта в 2010 году пришлось объяснять заказчикам, что модуль Excel 2003 для работы с OLAP-кубами, тоже от Microsoft, внезапно

¹ В оригинале звучит как «Don't put all your technical eggs in one basket».

был удалён с их веб-сайта, и сходную функциональность официальным путём теперь можно получить только в версиях 2007 и 2010.

Итогом истории для меня стал переход на Libre Office, где редактировалась в том числе и эта книга.

SQL Server

Несмотря на предвзято позитивное отношение к этому продукту Microsoft, используемому мною с 1996 года, вынужден отметить, что и здесь, начиная с 2005-й версии, наряду с полезными нововведениями проявилась определённая деградация.

Изменилась концепция пользовательского интерфейса. В версиях 7 и 2000 интегрированные приложения администратора (Enterprise Manager) и разработчика (Query Analyser) были разделены (в 6.x разделение тоже было, но неявное). В 2005 году корпорация решила унифицировать подход и свести всё в одну среду разработки и администрирования. Приложения переписали на .NET, благодаря чему они стали работать медленнее. В процессе переделки разработчики забыли, видимо, в спешке прихватить разные полезные мелочи, вроде множественного выделения объектов в дереве, которые за предыдущие годы были воплощены в версии 2000. Что, например, вы ожидаете увидеть, щелкнув мышью дважды на хранимой процедуре в правом окошке проводника объектов? Правильно, её текст. Но тут вас постигнет разочарование, мы проваливаемся в список параметров.

Если в версии 2000 пошаговая трассировка хранимых процедур и триггеров из консоли Query Analyser была доступна, то в 2005 она исчезла, а программистам порекомендовали дополнительно покупать Visual Studio соответствующей редакции. Хочется надеяться, что победил здравый смысл, а не поток жалоб, и в 2008 году отладка вновь появилась в основном инструментарии.

Поставляемая вместе с продуктом часть Visual Studio, видимая пользователю как BIDS (Business Intelligence Developer Studio), привязана к версии СУБД: OLAP-проекты, отчёты и пакеты ETL¹ для SQL Server 2005 могут разрабатываться только в BIDS 2005. А в BIDS 2008 уже не могут, потеряна обратная совместимость.

¹ От англ. Extract, Transform, Load — пакет для перекачки и преобразования данных между базами.

При этом многие полезные функции в 2005-й версии были всё ещё сырые. Например, `OPENQUERY()`, не допускающая конкатенации строк в параметрах, вынуждала писать трудносопровождаемый код внутри строковых переменных. Или новоиспечённая `ROW_NUMBER()`, не решающая, а создающая проблемы постраничной выборки на относительно больших объёмах. Исправляющая положение инструкция `ORDER BY OFFSET` появилась только в версии 2012. Форматы резервных копий между SQL Server 2008 и 2008 R2 оказались несовместимы.

Со средствами сетевого доступа к серверу баз данных и вовсе вышла смешная история.

В версии SQL Server 6.5, которая была ещё «почти Sybase 11», родным интерфейсом доступа к СУБД являлась DB-library в виде DLL и статической библиотеки C/C++, а драйвер ODBC¹ шёл как стандартное дополнение. После выпуска полностью переписанной на уровне ядра версии SQL Server 7 в 1998 году Microsoft анонсировала отказ от DB-library в пользу OLE DB, утверждая, что именно он и будет теперь «родным» для СУБД. Сама постановка фразы о том, что универсальный интерфейс, работающий через COM, может быть одновременно «родным», вызвала в сообществе разработчиков недоумение.

Поддержка DB-library была прекращена, в SQL Server 2000 эта библиотека ещё присутствует, но уже работает поверх (!) OLE DB. Библиотека C/C++ поверх COM — яркий пример костыля, подпирающего этаж совместимости. Имевший специфичные расширения ODBC-драйвер был сделан «с нуля», он самостоятельно реализовывал протокол TDS² доступа поверх сетевого уровня IP-сокетов и являлся высокопроизводительным и автономным, то есть не требовал установки дополнительных «родных» DLL. SQL QueryAnalyser в версии 2000 и, позднее, SQL Server Management Studio используют ODBC для управления запросами.

Одновременная поддержка OLE DB и ODBC в синхронном функциональном состоянии требовала затрат, которые в 2012 году решили сократить. Microsoft анонсировала отказ от OLE DB в пользу ODBC для работы с SQL Server.

Оказалось, что «родной» интерфейс всё-таки не родной. «Родного» у SQL Server просто нет, но есть ODBC, который теперь объявлен «родным»,

¹ От англ. Open Database Connectivity — программный интерфейс доступа к базам данных, разработанный фирмой Microsoft на основе спецификаций Call Level Interface (CLI). CLI был стандартизован ISO/IEC 9075-3 в 2003 году. Существуют реализации ODBC под UNIX, включая ODBC-драйвер для SQL Server под Linux.

² От англ. Tabular Data Stream — протокол прикладного уровня, реализующий обмен данными между СУБД и клиентом. Используется в продуктах Sybase и Microsoft.

и SQLClient для .NET. Могли ли в 1995 году немногочисленные разработчики, выбравшие тогда ODBC, предполагать, что к 2012 году они окажутся в числе выигравших?

Vista

Windows 7 ещё находилась в состоянии «кандидат к выпуску», а представители Microsoft открытым текстом стали предлагать отказаться от покупки своего флагманского на тот момент продукта — операционной системы Windows Vista и подождать выхода новой версии.

По сути пользователям сообщили, что мы достаточно потренировались на вас и за ваши же деньги, а теперь давайте перейдём собственно к делу. Судьба Vista была решена — система фактически выброшена в мусорную корзину, так и не успев занять сколь-нибудь значительную долю парка «персоналок» и ноутбуков. Производителям железа была дана отмашка переключиться на Windows 7, корпоративным службам ИТ пришлось в срочном порядке сворачивать проекты по переходу на Vista и ориентироваться на «семёрку».

Не скрою, я изначально был скептически настроен к Vista, вышедшей в конце ноября 2006 года, по двум причинам. Во-первых, ничего существенного по сравнению с Windows XP она не приносила, а во-вторых, первый пакет обновлений, после которого обычно можно начинать работу с продуктами Microsoft, задержали более чем на год, до марта 2008 года. Такие повороты неприятны не тем, что за ними обычно скрывается некий тайный смысл и прочая конспирология, а тем, что, судя по заявлениям для публики, стоит за всем этим нескоординированная работа разных служб и сумбурная техническая политика последних лет.

Интересно также, что Билл Гейтс и Стив Балмер в конце 2000-х годов начали активно продавать свои доли в бизнесе Microsoft.

О материальном

И немного о материальном, про оборудование. Периферия. Широко распространено мнение о том, что Linux на порядок хуже чем Windows поддерживает всякие периферийные устройства. Однако это не совсем так.

Если брать 32-разрядные версии Windows, то, действительно, нетрудно найти драйвер даже к довольно старому устройству. Но Microsoft уходит с 32-раз-

рядных платформ на десктопах и ноутбуках, навязывая предустановленные 64-разрядные версии своих операционных систем. Некоторые приложения всё чаще требуют 64-разрядной среды, например, для обработки больших объёмов данных или видео высокой чёткости.

Тут-то и выясняется, что по уровню поддержки периферии 64-разрядные Windows отрезают вам путь к использованию ещё совсем старых устройств. Если у вас в доме или на работе имеется оборудование 2–3-летней давности, то с большой вероятностью найти драйверы даже под Vista x64 вам не удастся. Не говоря уже о Windows 7. Единственный доступный выход — виртуальная машина с Windows XP.

А что же в Linux? Если оборудование поддерживалось в предыдущих версиях таких распространённых дистрибутивов, как Ubuntu или Mint, то независимо от разрядности новых версий системы оно будет продолжать поддерживаться и в них после обновления. Сканером и лазерным принтер я пользуюсь уже более пяти лет, сначала в 32-битной Windows XP, установив драйверы из комплекта, а затем под 64-разрядными Ubuntu и Mint из дистрибутива, не требующими установки драйверов вообще. Чего и вам желаю.

Проектирование и процессы

«La perfection est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à retirer».
(Совершенство достигается не тогда, когда нечего добавить, а тогда, когда нечего убрать.)

Антуан де Сент-Экзюпери

Корпоративные информационные системы (КИС) прошли долгий путь от полной закрытости сплавленных с аппаратурой монолитов до создания модульных и открытых систем. Однако теперь вместо стандартизации процессов и реализации лучших практик создания базовой функциональности на передний план выходят конкурентные преимущества за счёт дифференциации и специализации. Прежде всего, за счёт обрастания «скелета» КИС «мышцами и кожей» специфичных для данного предприятия программ.

В разделе собраны заметки и наблюдения, касающиеся архитектуры, некоторых практик концептуального и технического проектирования, реализации автоматизированных информационных систем и сопутствующих процессов.

Краткий словарь для начинающего проектировщика

Основная задача проектировщика — поиск простоты. Очень просто делать сложно, но очень сложно сделать просто. Начинающий проектировщик осознает это сам со временем, а пока нужно учиться элементарным понятиям для понимания, что же хотел сказать коллега по проекту на самом деле.

- «Это был плохой дизайн». Это спроектировано не мной.
- «By design» (так спроектировано). Ошибка проектирования, стоимость исправления которой уже сравнима с переделкой части системы.
- «Это не ошибка, а особенность (*not a bug but a feature*)». Прямое следствие из «by design».
- «Это может ухудшить производительность». Не знаю и знать не хочу ваши альтернативные решения.
- «Нормализация не догма». Потом разберёмся с этими базами данных, когда время будет.
- «Это наследуемый модуль». Этот кусок со многими неявными зависимостями проектировали достаточно давно, скорее всего стажёры.
- «Постановка задачи тоже сложна». Ума не приложу, откуда возникли эти десятки тысяч строк спагетти-кода.
- «Сроки очень сжатые». Мы давно забили болт на проектирование.
- «Наши модульные тесты покрывают почти 100 % кода». А функциональными тестами пусть занимается заказчик.
- «В нашей системе много компонентов». Установку и развёртывание системы могут сделать только сами разработчики.

Слоистость и уровни

Разбираться в слоях и уровнях должны не только разработчики КИС, то есть «скелета», но и те программисты, которые будут наращивать на него свои приложения. Иначе велик риск ненароком прилепить бицепс вместо ягодичных мышц или наоборот.

Определение автоматизированной информационной системы (АИС) складывается из трёх основных её компонентов: людей, информации и компьютеров. Любая АИС — это люди, использующие информационную технологию средствами автоматизации [9]. И КИС не исключение.

В публикациях по софтверостроению часто используются понятия слоёв и уровней программной системы. В англоязычной среде соответствующие термины —

layer и *level*. На физическом уровне реализующий слой компонент системы называется звеном — *tier*. Чтобы не запутаться в употреблении терминологии, нам следует чуть подробнее взглянуть во внутреннее устройство АИС.

Любая автоматизированная информационная система может быть рассмотрена с трёх точек зрения проектировщика:

- концептуальное устройство¹;
- логическое устройство;
- физическое устройство.

Концептуальное устройство

Концептуальное устройство АИС составляют всего три слоя.

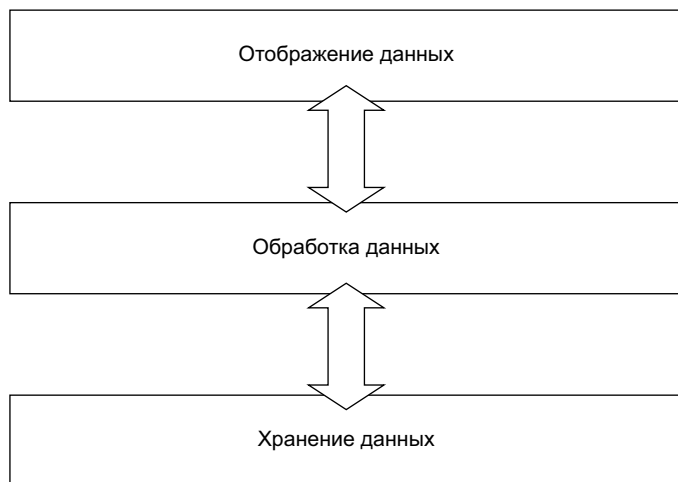


Рис. 4. Концептуальные слои АИС

Слои концептуального устройства существуют в любой, подчёркиваю — в любой, информационной системе, даже если с точки зрения физической архитектуры или конкретного программиста их трудно различить. Это тот

¹ Термин «устройство» в данном контексте примерно соответствует другим часто употребляемым терминам: «архитектура», «дизайн», «концепт».

случай, когда незнание закона не освобождает от ответственности. Поэтому декомпозиция системы на концептуальные слои является предметом анализа, а не синтеза. Результатом этапа (или этапов) анализа являются, соответственно, концептуальные модели данных, их обработки и ввода/вывода, включая человеко-машинные интерфейсы.

Логическое устройство

Напротив, слои логической архитектуры не являются строго определёнными. Основным способом их выделения является постоянный диалог проектировщика с требованиями к системе и ответами на вопрос «Зачем нужен этот слой в данном случае?». Наиболее типовые вопросы и ответы сведены в так называемые шаблонные решения и рекомендуемые практики.

Логическое устройство является предметом синтеза, на выходе стадии — технический проект. Логическое устройство АИС в разрезе концептуальных слоёв может выглядеть, как на рис. 5.

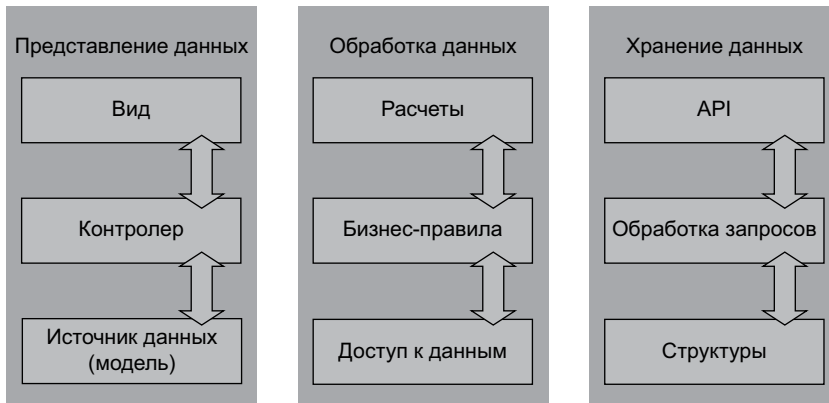


Рис. 5. Пример организации логических слоёв АИС

Физическое устройство

Аналогично логической архитектуре, физическое устройство тоже является предметом синтеза на стадии проектирования реализации. Физический слой также называется звеном (*tier*).

Число звеньев системы определяется **максимальным** количеством процессов клиент-серверной архитектуры, составляющих цепочку между концептуальными слоями.

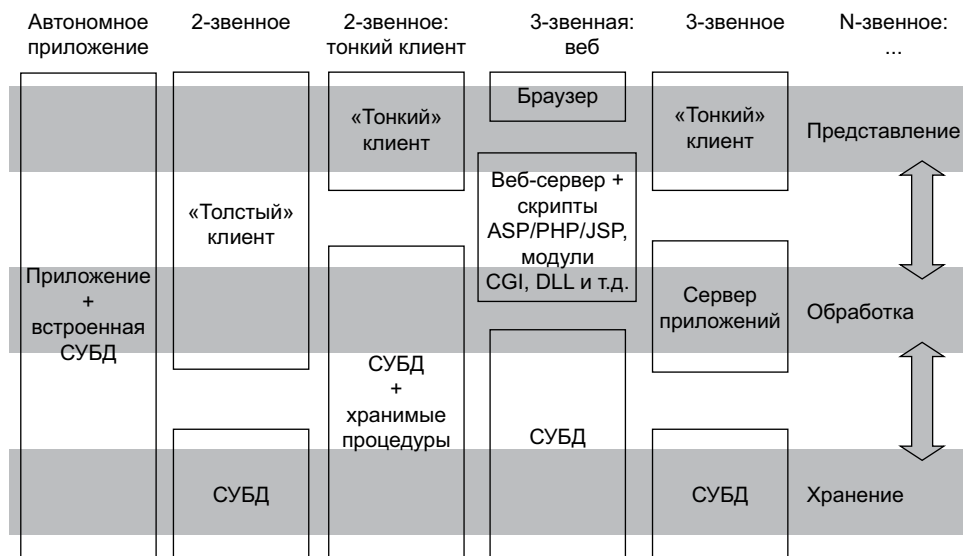


Рис. 6. Пример организации звеньев АИС

Тонким клиентом (*thin client*) традиционно называют приложение, реализующее исключительно логику отображения информации. В классическом варианте это алфавитно-цифровой терминал, в более современном — веб-браузер.

В противоположность тонкому, толстый клиент (*rich client*) реализует прикладную логику обработки данных независимо от сервера. Это автономное приложение, использующее сервер в качестве хранилища данных. В трёхзвенной архитектуре толстым клиентом по отношению к СУБД является сервер приложений.

Так называемый «умный» (*smart client*) клиент по сути остаётся промежуточным решением между тонким и толстым собратьями. Будучи потенциально готовым к работе в режиме отсоединения от сервера, он кэширует данные, берет на себя необходимую часть обработки и максимально использует возможности операционной среды для отображения информации.

Не секрет, что возможности отображения у веб-браузера, как программируемого терминала, очень скромные, по сравнению с автономным приложением. Компромиссным решением является так называемое «насыщенное интернет-приложение»¹, также являющееся тонким клиентом, но обладающее всеми возможностями отображения клиента толстого.

Уровни

Даже в простой программе типа записной книжки имеется минимум 2 уровня:

- Уровень приложения, реализующий функционал предметной области.
- Уровень служб, поддерживающих общую для всех разрабатываемых приложений функциональность. Например, метаданные, безопасность, конфигурация, доступ к данным и т. д.

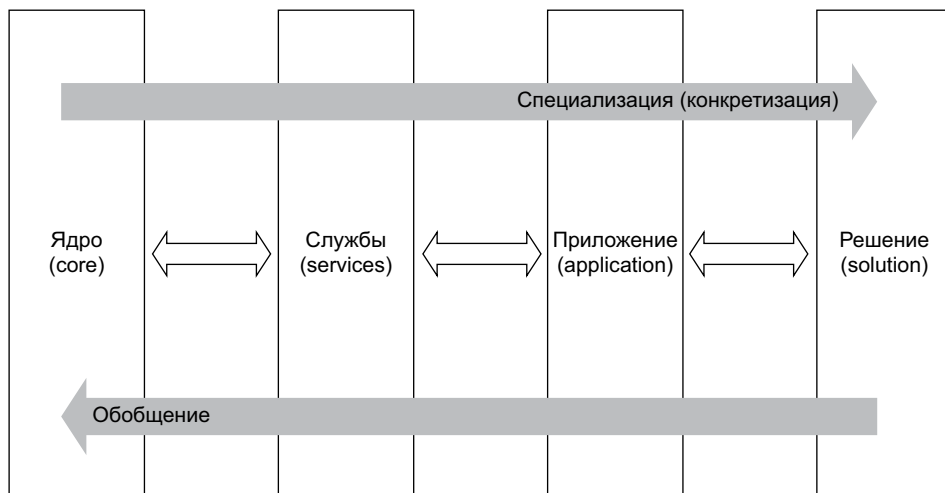


Рис. 7. Уровни АИС

В свою очередь, общие для многочисленных служб функции группируются в модули и соответствующие API уровня ядра системы. Таковы, например, API оконной системы графического интерфейса, SQL для доступа к реляционным

¹ От англ. rich internet application (RIA).

базам данным, основанные на HTTP-протоколах веб-служб или компонентная среда сервера приложений.

С другой стороны, комплексная информационная система обладает множеством приложений, реализующих функционал нескольких предметных областей. В этом случае функциональность, связанная с их интеграцией и управлением, поднимается на уровень решения. Например, конфигурация информационных потоков между приложениями, включая пакетную обработку.

Совмещение

Теперь если мы наложим слои системы на её уровни, то получим достаточно простую матричную структуру, позволяющую бегло оценить, какой из элементов необходимо реализовать своими силами или же адаптировать уже имеющийся готовый. Но, что более существенно, реализация разных подсистем может осуществляться независимо друг от друга после спецификации своих межуровневых и межслойных интерфейсов.



Рис. 8. Уровни АИС в разрезе концептуальных слоёв

Многозвенная архитектура

Итак, концептуальных слоёв в автоматизированной информационной системе всегда три: хранения данных, их обработки и отображения. А вот физических,

их реализующих, может быть от одного, в виде настольного приложения с индексированным файловым хранилищем, до теоретической бесконечности.

Имея опыт разработки систем всех перечисленных на рис. 6 типов, наиболее позитивные впечатления у меня остались от «двухзвенки» с тонким клиентом. Подробнее я расскажу об этой архитектуре в главе про разработку тиражируемой КИС Ultima-Seller.

Разумеется, у каждой архитектуры есть свои преимущества и недостатки. Если подходить к вопросу проектирования объективно, то выбор в каждом конкретном случае вполне рационален. Но я вспоминаю, например, что в начале профессиональной деятельности нам хотелось просто попробовать «сделать трёхзвенку» своими руками, невзирая на то, что экономическая целесообразность такого выбора была не совсем очевидна как по затратам на разработку, так и по стоимости владения системой в будущем.

Сегодня доля специалистов, имеющих опыт в системном проектировании, в общем потоке становится всё меньше. Поэтому декомпозиция и выбор архитектуры часто проводится по принципу «прочитали статью, у этих парней получилось, сделаем и мы так же». Огород из нескольких физических уровней насаждается не потому, что это действительно надо, а потому, что очередной гуру написал об этом в своём блоге. Или потому, что нет другого опыта и мотивации его приобрести: чему научили ремесленника, тому и быть.

В итоге между экраном конечных пользователей и запрашиваемыми ими данными образуется толстая прослойка, которая на 80 % занята совершенно пустой работой по перекачиванию исходной информации из одного формата представления в другой. Регулярные восстановления долговременных объектов, бесконечные сериализации и десериализации, передача и преобразование XML, дёрганье за веб-службы... С учётом, например, обновления области экрана по изменению каких-то данных в другой его части эти мытарства умножаются в разы. Растёт время отклика системы. Пользователь нервничает и справедливо обвиняет в этом программистов.

В такой ситуации нет ничего необычного, потому что практика управления «по кейсам»¹, то есть прецедентам, является широко распространённой в среде менеджеров среднего звена. Критичность основной массы проектов в ИТ снижается, соответственно, большая часть решений переходит из технической плоскости в управленческую. Эта тенденция будет только усиливаться.

¹ От англ. case — пример успешного проекта в терминологии управленцев.

Любопытно, что в русском языке слово «менеджер» имеет весьма точный, но основательно подзабытый в эпоху советской России перевод «приказчик». Магическая сила слов! Получил бы популярность клон игры «Монополия», если бы его в конце 1980-х годов назвали «Приказчик»? Теперь сравните пару «менеджер проекта» и «тестировщик» против другой: «приказчик» и «инженер-испытатель»...

Кто не хочет — ищет причины, кто хочет — средства. Ищите возможности сократить путь информации от источника к пользователю и обратно. В конце концов, архитектура служит для эффективной реализации системы, а не освоения бюджета и вовлечения в команду очередных выпускников курсов переквалификации. Подход разработки «по модели», о котором мы ещё поговорим, позволяет генерировать многие слои без участия программистов.

История нескольких `#ifdef`

640 килобайт памяти должно
хватить каждому.

Билл Гейтс (приписывается), 1980-е годы

Перелом начала 1990-х годов в России вынудил целые коллективы квалифицированных системно мыслящих людей переходить из специализированных учреждений НИОКР непосредственно на уцелевшие производства, эмигрировать или начинать собственный бизнес. Не мудрено, что в такой ситуации уровень технической культуры в отделе программирования торгово-производственной компьютерной фирмы «Ниеншанц» был достаточным и для софтостроительных проектов. Разумеется, не последняя роль в создании атмосферы и соответствующего интеллектуального уровня принадлежала основателям компании, пришедшим в бизнес из научной среды: Виктору Ярутову, Дмитрию Разгуляеву, Егору Макарову — выпускникам физического факультета ЛГУ¹.

¹ Ленинградский Государственный Университет.

Таким образом, уже в начале—середине 1990-х на предприятии существовала собственная комплексная информационная система Seller 2, выполненная в трёхзвенной архитектуре с технологией если и не совсем «умного», то и не вполне толстого клиента. Хотя разработчики ещё не знали термина *smart client*, это не помешало им аналогичным образом реализовать многие элементы подсистемы представления.

В следующей главе речь пойдёт о разработке тиражной КИС в «Ниеншанце». Разумеется, возникла она не на пустом месте. Чтобы самому не пересказывать предысторию появления нового продукта в виде предшествующих версий, я обратился непосредственно к участникам, своим бывшим коллегам. С небольшими поправками текст публикуется с их согласия.

То, что рассказывают ведущие программисты Дмитрий Цуранов (ДЦ), Сергей Быков (СБ) и Игорь Паньшин (ИП), кандидат технических наук, участвовавший в разработке на стадии платформы VAX¹, вначале было весьма близко к современным понятиям «эволюционный дизайн» и «гибкие методики». Но потом приходит осознание и понимание.

Начало

ИП: Все начиналось на Рижском проспекте (бывший пр. Огородникова), 26 в Институте Аналитического Приборостроения при Академии Наук СССР. Основной нашей деятельностью была разработка программного обеспечения вообще. Нас кидало из стороны в сторону. От разработки графики для терминалов Radiance, отладки кластеров VAX 6320 и VAX station на базе сети DECnet, редакторов, пакетов цифровой обработки сигналов до баз данных Datatrieve, с которыми можно было общаться в командной строке, каких-то игр и просто удовлетворения интереса работой типа «псевдоклавиатурный драйвер». Тогда, на излёте СССР, я постоянно искал дополнительные заработки. Среди моих знакомых был Д. Разгуляев, уже занимавшимся бизнесом, результатом которого стала фирма «Ниеншанц», располагавшаяся в то время около станции метро «Чернышевская».

ДЦ: Около 1991 года мы в «Аналитприборе» начали делать систему по заказу фирмы «Ниеншанц». Какую систему? Трудно сказать. Первое техническое задание в виде клочка бумажки с закорючками родилось значительно позже,

¹ VAX — 32-разрядная архитектура ЭВМ фирмы DEC, разработка середины 1970-х годов. На базе DEC PDP-11 в СССР была создана линия СМ ЭВМ.

а тогда... Тогда мы делали систему, которая работает с базой данных. Не больше и не меньше.

В качестве базы использовалась СУБД VAX RDBMS, а само программирование шло на голом С. Было очевидно, что понадобится прокрутка данных, для чего напрашивались курсоры. Как выяснилось, имеющиеся в VAX курсоры блокировали данные, в результате чего было принято решение отказаться от них вообще, соорудив структуру, называвшуюся «вектор». Вектор — это результат выборки данных, по которой перемещается текущая позиция. Когда вы уходите вверх или вниз, подгружаются новые данные, не одна строчка, а сразу пакет, чтобы обращения к базе были реже. Библиотечка на С, предоставлявшая интерфейс к векторам, получила гордое название CST, или Client Server Tool.

СБ: Первоначальная архитектура системы была экзотической — сервер VAX с RDBMS, клиент на ПК. Она была обусловлена тем, что у «Ниеншанца» уже был один VAX. В числе прочих, рассматривался вариант с тонким клиентом: база данных VAX RDBMS, приложение на С или Паскале под VAX/VMS и тонкий клиент на ПК по протоколу X11. Несколько месяцев мы экспериментировали с X Windows.

#ifdef NWSQL (1991–92 год)

ДЦ: Спокойное время в «Аналитприборе», безвозмездно, то есть даром, предоставлявшем нам VAX, кончилось, и мы перешли в фирму «Ниеншанц», где столкнулись с персоналками IBM PC и операционной системой MS-DOS. Впечатление, которое произвела на нас однозадачная MS-DOS после 32-рядной VAX/VMS с *preemptive multitasking*¹ и защищёнными адресными пространствами, было гнетущим. Но там платили деньги.

В качестве сервера использовался Novell NetWare, а в качестве базы — NetWare SQL, и по этому случаю код библиотечки «векторов» пополнился многочисленными `#ifdef NWSQL`. Так как мы озаботились целостностью данных, то на Watcom C был написан серверный модуль NLM², обеспечивавший

¹ Вытесняющая многозадачность — в отличие от корпоративной многозадачности, приложение не способно монополизировать ресурсы операционной системы, поэтому может быть временно приостановлено, а «зависшие» приложения не блокируют остальные.

² NLM — NetWare Loadable Module — формат серверного приложения под Novell NetWare. До 4-й версии NetWare работали только в нулевом кольце процессов, то есть при сбое могли вызвать крах всей системы.

механизм пессимистичных блокировок и даже рассылавший сообщения о модификации данных, что проводило к автоматическому обновлению векторов.

Кроме того, у нас появился программист, отвечающий за GUI¹. Он разрабатывал довольно своеобразный редактор форм. Впрочем, у нас все было своеобразным.

СБ: «Ниеншанц» решил, что VAX — это слишком дорого для собственной КИС. Вот тогда возникли Novell и Btrieve, которые были бесплатным к нему приложением.

Новым программистом GUI был Юрий Дымов по прозвищу «папа», потому что, даже будучи младше нас, в 1992 году он уже был женат, имел дочь. Юра обладал богатым арсеналом приёмов программирования, о котором говорит тот факт, что один раз утечку памяти он пытался исправить сменой компилятора С. Он написал собственный менеджер памяти, то, что современным языком называется *small memory heap*², и заставил конструкторы сторонней графической библиотеки работать через него. Без этого память у нас кончилась бы гораздо раньше...

ДЦ: Конечно, программисты такого типа не смущаются вставлять в код уродливые подпорки «чтобы работало». Они обычно плохо работают в длинных проектах, слишком много энтропии вносят в код. Зато если «кровь из носу» надо сделать так, чтобы работало сегодня к 16 часам, — они лучше всех.

ИП: В России наступала эра персональных ЭВМ, а я не мог бросить заниматься VAX-ами. Прямо, как чемодан без ручки. Правда, там были общедоступные исходники. Я имею в виду общество DECUS (Digital Equipment Corporation User Society). Поэтому пришлось сделать выбор.

#ifdef BTSQL (1992-93 год)

ДЦ: NetWare SQL был лишь надстройкой к СУБД Btrieve³, встроенной в Novell-овскую серверную операционную систему. Причём эта надстройка выполнялась на клиенте в специальной резидентной программе *brequest*. Она занимала 280 ки-

¹ Graphical User Interface — графический интерфейс пользователя, в данном контексте — терминальное приложение.

² Heap — в непосредственном переводе «куча», область динамически распределяемой во время выполнения программы памяти.

³ Btrieve не является полноценной СУБД, а относится к так называемым менеджерам записей.

лобайт. Современный человек не поймёт, о чем речь. Он занимал целых 280 из 640 килобайт! Плюс MS-DOS и драйверы, а в оставшихся 300 килобайтах как хочешь, так и крутись... Кроме того, почему-то начальство считало полезным отказаться от «платного» NetWare SQL, будто в России тогда за что-то платили.

В итоге был написан небольшой слой, который находился под CST и транслировал узкое подмножество SQL без соединений (*joins*) в запросы к Btrieve. Наверное, это худший код, который я написал за свою жизнь, потому что он целиком находился в файле `btsql.c` — пара тысяч строк на чистом C. От обилия глобальных флагов я впадал в панику и вводил новые, только чтобы не трогать старый код. Ну а векторы, как и положено, обросли `#ifdef BTSQL`.

Работа с Btrieve была ещё тем удовольствием, поскольку отсутствовало понятие логического поля. Например, при создании индекса указывалось, что индекс включает байты с 3-го по 9-й, а второй сегмент — с 22-го по 25-й. Список таблиц, полей, смещений к началу полей в записи приходилось вести самостоятельно.

Интересно, что Btrieve имела уникальный двухверсионный уровень изоляции, который я нигде больше не встречал. Процессы-читатели никогда не блокировались и не считывали «грязные»¹ данные, а если напарывались на них, то брали предыдущую чистую версию. Понятно, что какая-либо целостность этой версии по времени не гарантировалась: каждая таблица хранилась в своем файле. Журналов транзакций не было, «грязные» данные хранились в специальных страницах (*dirty pages*) в той же самой таблице.

СБ: Кроме первой пробной установки Novell, за все остальные компания честно платила. Поэтому стоимость лицензий NetWare SQL была серьёзным аргументом в дополнение к его слабому быстродействию. CST на «голом» Btrieve работал в разы быстрее.

Кроме системы учёта для «Ниеншанца», на CST в 1992 году была сделана система для Молодёжной Биржи Труда. Как минимум год мы её сопровождали. В том же году сервер CST демонстрировался на выставке в ЛенЭкспо.

NDL, или Java в миниатюре (1993–94 год)

ДЦ: Между тем система, построенная на всем перечисленном, уже активно использовалась в компании и назвалась Seller 1.0. Написана она была, кроме интерфейса, опять-таки на голом C, а из-за ограничения в 640 килобайт пред-

¹ В терминах СУБД «грязными» называются данные неподтверждённой транзакции. Считав их, вы не можете знать, сохранится ли новое значение или вернётся к предыдущему.

ставляла несколько разных исполняемых модулей: `pay.exe` для бухгалтерии, `seller.exe` для продавцов, `store.exe` для склада... Универсальную программу по причине размера собрать было уже нельзя.

Однако ограниченность платформы была ясна, и мы предприняли попытку разработать свой, как говорят сейчас, фреймворк. Да, мы создали в кратчайшие сроки свой язык NDL¹, компилятор, компоновщик и исполняющую систему, независимую от ОС. Она была переносима без каких-либо серьёзных проблем, в переходный период код исполнялся одновременно и под DOS, и под Windows. В ней была даже реализована бесконечнозначная арифметика с фиксированной точкой, хотя в самом NDL количество знаков ограничивалось 64.

Но Windows будет потом, а пока исполняющая система под DOS научилась грузить NDL-программу в расширенную оперативную память (*extended memory*), освобождая под данные почти всю память основных 640 Кбайт минус исполняющая система, это был грандиозный прорыв. Кроме того, в NDL были исключения (*exceptions*) и куча полезных функций для работы именно с базами данных, так что мы бодро принялись писать новую систему Seller 2 на ней.

СБ: Если Seller 1 делался по наитию Д. Разгуляева, уточнявшего каждую неделю техзадание, то Seller 2 мы делали «правильно». Месяца 2–3 мы практически «бездельничали» — то есть придумали и обсудили приличное количество идей, оставив то, что нам казалось лучшим. Нарисовали схему базы данных и даже составили список ключевых функций. И только после этого приступили к реализации.

Сам NDL начинался с интерпретатора. Был придуман мета-код, первые бизнес-функции писались прямо на нем, но через месяц стало ясно, что без нормального языка дальше жить нельзя. Тогда появился NDL, компилятор и компоновщик. За основу синтаксиса был взят Паскаль. Код компилятора генерировался по описанию грамматики конвейером из 2 утилит, `lex` и `yacc`, под FreeBSD, установленной на одном из серверов. Полученный код на C затем компилировался в среде Borland под Windows. Компоновщик NDL собирал проект по описанию в `makefile` с разрешением имён глобальных и локальных ссылок и объектов. При этом он мог оставлять комментарии, которые позволяли исполнять код в режиме отладки с позиционированием на строки исходного кода. Всё было по-взрослому, несмотря на то, что сделано командой из 3 человек менее чем за год.

¹ Network Data Language, позднее Nienschanz Development Language.

Закат Novell

ДЦ: Начиналось все хорошо. Вышел новый Btrieve версии 6, где появились интерактивное резервное копирование (*online backups*) и поиск без индексов. В пятой версии можно было либо читать все данные, либо искать по индексу. А для проверки типа «Сумма=123» необходимо проверять записи на клиенте. Теперь можно было эти проверки перенести на сервер.

Поначалу я этому обрадовался: задал сложный поиск, и сервер думает минуту. Пока не заглянул на заднюю панель компьютера. Лампочки, показывающие активность сетевой карты непрерывно горели! Маленький резидент на клиенте непрерывно посылал запросы типа «Готово? — Ещё нет». Иначе он и не мог, ведь раньше все обращения к Btrieve были короткими, и ждать не предполагалось...

Но тут вышла Windows 95 под кодовым названием «Чикаго». Novell почему-то не торопилась делать 32-разрядные драйверы и вообще как-то реагировать на новый тренд. В итоге Microsoft сама сделала 32-битные версии драйверов для IPX/SPX, но воспользоваться ими было невозможно, так как пресловутый brequest работал в 16-разрядном режиме DOS.

Наконец, я нашёл файл с говорящим именем `breq32.dll` (в «догугловую» эпоху это было делом дней и недель, а не секунд) и... выяснилось, что она представляет собой лишь 32-разрядный интерфейс для обращения к пресловутому brequest.

Последний гвоздь в крышку гроба Novell, как платформы разработки, был забит с попытки запустить систему под Windows NT. Все заработало правильно, но раз в 10 медленнее. Так у нас появился следующий `#ifdef`.

СБ: У Novell был лучший файловый сервис, который я видел. Нормальное управление правами доступа, *garbage* — мусорная корзина, была реализована на уровне ядра, никаких проблем с восстановлением неосторожно удалённых файлов. Поддержка RAID. Но всё остальное, что делала компания, было ужасно. Их связка Btrieve + NetWare SQL была тупиком. Они это понимали и родили кентавра OracleWare — NetWare со встроенной СУБД Oracle. Мы были на презентации этого чуда в отеле «Европа».

Сегодня я считаю, что такой альянс был ошибкой, так как Novell базировался на архитектуре x86. Тем, кому был нужен Oracle, такие решения не подходили, они в 1990-х покупали серверы на многопроцессорных RISC-архитектурах, например SPARC, а Novell был продуктом более низкой ценовой категории.

Думаю, они специально не торопились с драйверами под Windows и поддержкой доменов NT в своей службе каталогов, понимая, что Microsoft — их прямой конкурент, поскольку Windows объединила в себе функции сетевой

серверной и настольной ОС. Vtrieve под NT появился в результате выделения продукта в отдельную компанию.

От автора: Я снова столкнулся с Novell в 2010 году в рамках небольшого проекта для французской национальной сети телевидения. Корпоративная система безопасности для тысяч компьютеров на разных территориальных площадках была по-прежнему построена на службе каталогов NetWare, хотя и в тесной интеграции с аналогичной службой Microsoft. Новые компьютеры с Windows Vista/7 включались в общую систему. Сама Novell в конце того же 2010 года была куплена малоизвестной компанией Attachmate за целых 2,2 миллиарда долларов и формально прекратила существование. По некоторым сведениям, за Attachmate стояла Microsoft, незадолго до того выложившая 450 миллионов на приобретение у Novell технологий.

`#ifdef` Windows

ДЦ: Между тем мы стали переходить под Windows. Как уже говорилось, для кода бизнес-логики, написанной на NDL, переделок не потребовалось вовсе. Клиентское приложение было переделано, но понимало описания форм, сделанных ещё под DOS. Конечно, под Windows моноширинные шрифты и формы выглядели довольно уродливо, но тем же страдал и SAP R/3.

Пришлось нам переделывать и систему печати документов под лазерные принтеры, но это отдельная маленькая история. А вот что сильно портило настроение, это старая DOS-подсистема для Vtrieve. Стоило программе обратиться с запросом на сервер, и она уходила в себя. Ещё один камень в огород Novell. Поэтому вскоре у нас появился последний в развитии второй версии `#ifdef`.

СБ: Когда в Seller 2 появился небольшой модуль кадрового учёта, то при запуске система стала поздравлять пользователя с днём рождения. На 8 марта мы рисовали стартовый экран с поздравлениями в стихах. Один год на первое апреля мы перевернули драйвер мыши: двинешь мышь влево — курсор идёт вправо, мышь от себя — курсор вниз. В другой раз перевернули экран вверх ногами. Сейчас мне трудно представить, чтобы с утра у всех в конторе, включая директоров, появились перевёрнутые экраны и программистам за это ничего не было.

Также существовал файл настроек, управлявший цветовой гаммой интерфейса. Кто-то из нас проговорился, и пользователи про него узнали. Такой вакханалии цветов, как в отделе оптовых продаж, я не видел нигде...

#ifdef MSSQL

ДЦ: Система стала полностью 32-разрядной, перепрыгнула на полноценную СУБД. Интересно, что в коде можно было по-прежнему встретить `#ifdef VAXVMS...`¹

NDL переводил наиболее частые сообщения SQL на русский, чтобы пользователи не пугались. Приложение выдавало «табличку» — окно с сообщением на английском.

СБ: «Табличка» — термин одной из работниц склада.

— Валя, почему это табличка? Это же окно сообщения...

— Оно же в рамочке!

ДЦ: До боли знакомое всем работающим с SQL Server сообщение *«your process has been chosen as deadlock victim»* было переведено как «Вам дорогу перебежала черная кошка».

СБ: Сервер блокировок перестал использоваться после перехода на SQL Server 6.5 и отказа «Ниеншанца» от использования Novell в качестве файлового сервера. Помню, как одна из девочек-продавцов с удивлением узнала, что теперь удалённый файл нельзя восстановить, в корзине сервера его не было. Серверу блокировок стало негде работать, он был реализован в виде NLM, и его заменили на использование функции `app_lock`. И снова абсолютно прозрачно для бизнес-приложений — просто заменили в NDL реализацию соответствующей мета-команды.

Постскрипtum

Если вы дочитали предысторию до конца, вдумайтесь в цифры. КИС, реализующая основные функции автоматизации деятельности торгово-производственной фирмы среднего размера: от бухгалтерии и складов до сборочного производства и сбыта — была разработана командой из 4–5 человек примерно за полтора года, включая миграцию с предыдущей версии. Система критичная, даже короткий простой оборачивается параличом деятельности фирмы.

Причина столь сжатых сроков? Ясное понимание решаемых прикладных задач, создание соответствующего задаче инструментария, прежде всего, языка

¹ Интересно, что если каждую букву аббревиатуры VMS заменить на следующую по алфавиту, то получится WNT, то есть Windows NT. VMS — операционная система компьютеров фирмы DEC, специалисты которой в начале 1990-х годов участвовали в разработке ядра Windows NT.

бизнес-правил высокого уровня, и подтверждение тезиса Брукса о многократно превосходящей производительности хороших программистов по сравнению с остальными.

Последние годы в ходе аудита баз данных я не раз наблюдал, как современные команды в 2–3 раза большей численности, вооружённые умопомрачительными средствами рефакторинга и организованными процедурами гибкой разработки, за год не могли родить работоспособный заказной проект, решающий несколько специфичных для предприятия задач. Сотни тысяч строк кода уходили в мусорную корзину или продолжали поддерживаться с большими трудозатратами, сравнимыми с переделкой.

В другом случае четыре с половиной программиста сумели в короткие сроки создать и в течение многих лет сопровождать КИС для французского туроператора национального (один из крупнейших) и европейского уровня. Это уже потом к ней приделали веб-интерфейс для заказов клиентов и B2B¹-шлюз с партнёрами.

Есть о чём призадуматься, особенно желающим начать новый проект.

Ultima-S — КИС из коробки

На дворе стоял 1996 год, падение экономики если уже не прекратилось, то сильно замедлилось, но компания «Ниеншанц» приступила не просто к разработке очередной, третьей версии внутренней системы управления предприятием, а к созданию тиражируемого коробочного продукта. Словно в подтверждение мысли М. Донского о том, что техническая культура — это не производства и знания, а люди, умеющие это делать и применять [11].

Название у нового продукта возникло не сразу. К концу первого полугодия разработки продукта с кодовым названием Seller 3 руководство решило, что для широкого круга потенциальных клиентов название должно быть более «брендовым». Так появилась Ultima-S, где буква «S» перешла по наследству в качестве инициала прежнего имени.

Существует два основных подхода к разработке КИС, условно называемых «от производства» и «от бухгалтерии».

¹ B2B — от англ. business-to-business, шлюз между корпоративными информационными системами.

В первом случае функциональным ядром системы становится планирование ресурсов производства. Упрощенно: на предприятии есть сотрудники, оборудование и сырьё (материалы, компоненты), с одной стороны, а с другой — план выпуска — «чего и сколько?» вкупе с технологией — «как?», то есть правилами, нормами и прочими ограничениями процесса преобразования сырья в готовую продукцию. В первом приближении, необходимо составить оптимальный по загрузке персонала и оборудования план этого процесса. После того как система научилась составлять производственный план, она тянет за собой все остальные функции: от кадров, ведь персонал не из воздуха появляется, и снабжения сырьём до складирования и сбыта готовой продукции.

Альтернативный путь проходит через бухгалтерию. Под термином «бухгалтерия» имеется в виду прежде всего внутренний, управленческий учёт на предприятии, а не фискальная её часть. Дело в том, что механизмы бухучёта придумали ещё в XV веке вовсе не ради подачи отчётности в средневековую налоговую инспекцию, а для понимания состояния дел на своём предприятии. Бухгалтерский учёт — хорошо формализуемая аппаратом матричной алгебры[13] абстракция для отражения и анализа хозяйственных операций в дискретных периодах времени, в том числе и будущих.

Большинство известных мне разработок КИС в России 1990-х годов шло «от бухгалтерии». Причина достаточно простая. Производство за первые 5 лет упало более чем вдвое, при этом у заводов, как правило, уже имелись свои системы, в том числе перенесённые с мейнфреймов на «персоналки» собственными силами отделов программистов, чаще всего в рамках файл-серверной технологии. В то же время рос сектор услуг, оптовой торговли и дистрибуции, многие предприятия создавались «с нуля», и для их автоматизации производственные системы не подходили за отсутствием собственно производства.

Поскольку основными потенциальными клиентами Ultima-S были именно оптово-розничные торговые компании, то функциональная архитектура системы базировалась на подходе «от бухгалтерии».

Запустив в эксплуатацию и получив в сопровождение систему, кратко описанную в предыдущей главе, программисты на собственном опыте убедились, что каждый физический слой увеличивает трудоёмкость разработки, тестирования и последующих модификаций системы. Поэтому, с учётом предполагаемого тиражирования, развивать продукт было решено в следующих направлениях:

- минимизация числа звеньев;

- «уточнение» клиентского приложения, в идеале до уровня веб-браузера;
- использование промышленной СУБД для реализации бизнес-логики;
- реализация некоторых механизмов ООП для упрощения разработки прикладными и сторонними разработчиками методом надстраивания новых классов.

Синтезом вышеназванных приоритетов явилась двухзвенная архитектура с тонким клиентом.

Превратить промышленную СУБД в сервер приложений с технологической точки зрения просто. Для этого вам необходимо:

- запретить прямой доступ к таблицам базы данных;
- реализовать прикладную логику в виде хранимых процедур, функций и триггеров;
- разрешить доступ всех приложений только к соответствующим хранимым процедурам.

В технологии с тонким клиентом дополнительно потребуется:

- разработать протокол прикладного уровня для взаимодействия тонкого клиента и сервера приложений;
- запретить доступ ко всем объектам базы данных вообще, за исключением нескольких реализующих этот протокол хранимых процедур и, возможно, буферных таблиц.

Наконец, для надстройки над процедурным расширением SQL объектно-ориентированной среды, управляющей объектами предметной области, понадобилось:

- добавить поддержку декларации классов на уровне метаданных;
- реализовать механизм обработки сообщений между объектами;

- разработать набор базовых классов уровня ядра и служб системы (см. уровни).

Я не буду подробно останавливаться на сравнении преимуществ и недостатков использованной в продукте архитектуры, они достаточно известны и не раз обсуждались в разных формах. Скажу лишь, что для нас сумма преимуществ тогда перевесила. Во многих случаях перевесит и сейчас, даже если добавить ещё одно промежуточное звено из простейшей веб-службы, занимающейся ретрансляцией сообщений между терминалом и СУБД.

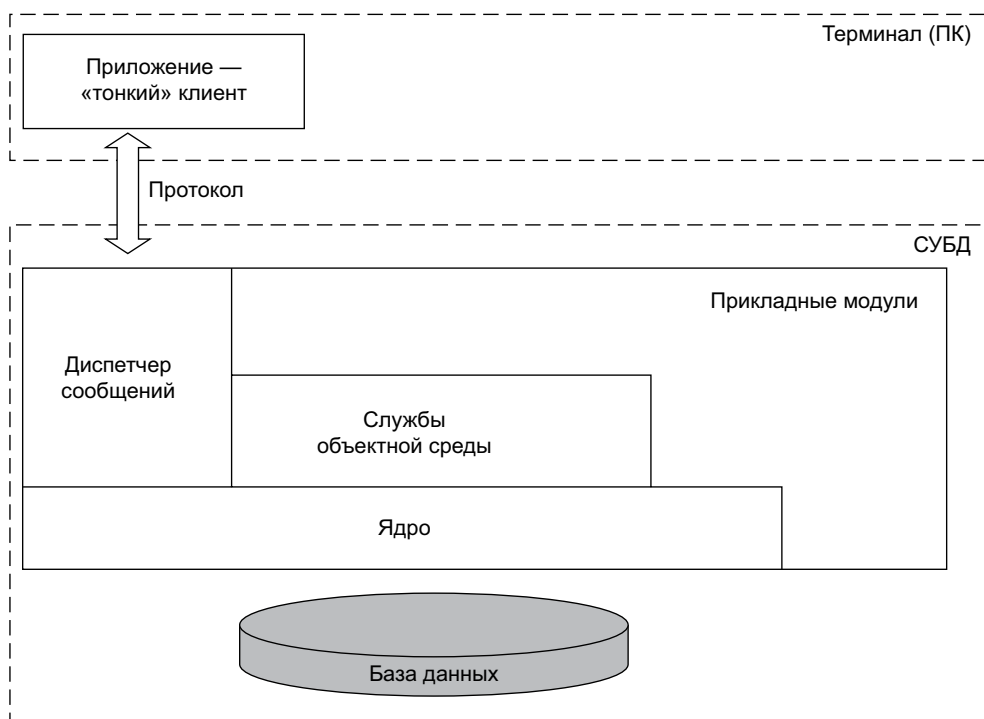


Рис. 9. Элементы логического устройства звеньев системы Ultima-S

В качестве базовой абстракции был выбран документ. То есть все объекты в системе — это документы, относящиеся к какому-либо их классу. Каждый документ хранился в одной из папок, составляющих иерархию, напоминающую вид обычного проводника Windows.

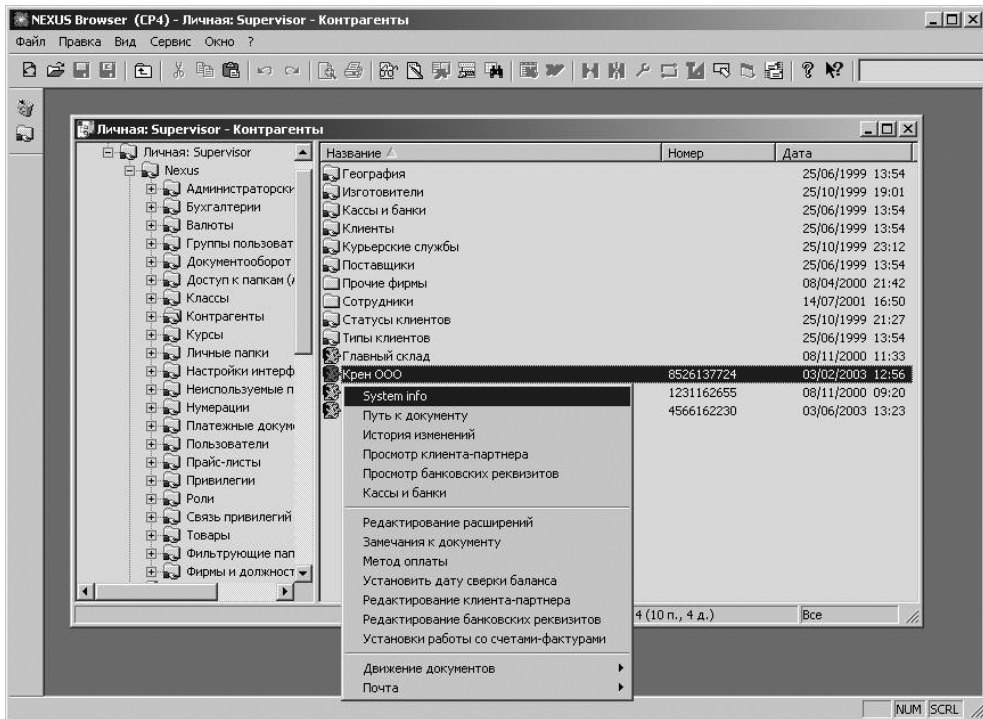


Рис. 10. Внешний вид тонкого клиента в КИС Ultima-S

В рамках механизма ООП, надстроенного над процедурным, поддерживалось одиночное наследование реализации. Вместо формальных конструкторов, деструкторов и методов основу составил механизм событий, вызывающий процедуры-обработчики в порядке, зависящем от иерархии классов.

В наибольшем выигрыше оказалось прикладное программирование. Сбылась «голубая мечта» — вся... нет, не так, ВСЯ разработка сосредоточилась в одном звене и среде на декларациях классов, свойств-операций и на реализующих обработку событий хранимых процедурах.

Тонкий клиент отображал в динамике результаты обработки события сервером. Большинство экранных форм, таким образом, формировалось автоматически, программист только объявлял в процедуре соответствующие поля ввода и сетки. Для специфичных случаев расположения элементов управления было необходимо визуально проектировать форму либо во встроенном в приложение редакторе, либо в среде Visual Basic, загрузив затем описание формы на сервер.

В общем случае, для создания программистом новых классов в системе было необходимо:

- декларировать классы;
- создать соответствующие таблицы;
- описать возможные ограничения на уровне метаданных (например, папку для создания по умолчанию или максимальное число ссылок);
- написать обработчики стандартных событий;
- добавить привилегии, видимые администратору;
- если необходимо, инициализировать данные, например, создать служебные объекты или документы-классификаторы.

Всё перечисленное программист мог сделать на макрорасширении над языком Transact SQL, не выходя за рамки разработки соответствующих скриптов. Приведу примеры использовавшегося кода.

Пример фрагментов кода модуля учёта сотрудников

```
-- Создаем таблицу для информации о сотруднике
CREATE TABLE Staff
(
  UDN      int          NOT NULL PRIMARY KEY,
  Gender   smallint NOT NULL,
  Chief    int          NULL,
  Deputy   int          NULL,
  Firm     int          NULL,
  Pos      int          NULL,
  Dept     int          NULL,
  Note     varchar(80)  NULL
)
GO

-- Декларация класса
EXEC ObjCreateClass 'Staff', 'Сотрудник', 'Doc', 0, 0, 0, 0, 0

-- Обработчик события getp (запрос списка операций с документом)
-- Операции отображаются в контекстном меню документа
CREATE PROCEDURE Staff_getp_edit_
  @p1 integer, @p2 integer, @p3 integer
AS BEGIN
  BeginProperties
  ViewProperty 'view', 'Просмотр'
  EditProperty 'edit', 'Редактирование'
END
```

GO

```
-- В обработчике события gets (запрос на создание нового документа)
-- определяем элементы динамически отображаемой формы ввода
CREATE PROCEDURE Staff_gets_edit_
    @p1 integer, @p2 integer, @p3 integer
AS BEGIN
    BeginDetail
    WindowTitle('Новый сотрудник')
    EditString 'Name', 'sФИО', ''
    EditInt    'gender', 'o0^Пол^Мужской^Женский', 0
    EditInt    'cheif', 'dРуководитель', 0
    EditInt    'deputy', 'dЗаместитель^home=Users^cl=Staff^def=0', 0
    EditInt    'firm', 'dФирма', 0
    EditInt    'pos', 'dДолжность^home=Pos^cl=Pos^def=0', 0
    -- Cycle – редактирование с запросом серверу и обновлением других полей формы
    CyclInt    'dept', 'dОтдел^home=Fstruct^cl=Fstruct^def=0', 0
    EditString 'note', 'sПримечания^len=128', 0
END
GO
```

```
-- В обработчике события cre (сохранение нового документа)
-- сохраняем введенные данные
CREATE PROCEDURE Staff_cre_edit_
    @p1 integer, @p2 integer, @p3 integer
AS BEGIN
    DECLARE @Name varchar(255)
    VarString(@Name; 'Name')
    IF EmptyString(@Name) BEGIN
        rerror('Не введено ФИО сотрудника')
        RETURN 1
    END

    -- ФИО записываем в таблицу базового класса Docs
    UPDATE Docs SET Name = @Name WHERE UDN = @p2
    -- Заполняем новую запись данными из формы
    INSERT INTO Staff (UDN, Gender, Deputy, Pos, Dept, Note)
    SELECT @p2,
        GetInt('gender'),
        GetInt('deputy'),
        GetInt('pos'),
        GetInt('dept'),
        GetString('note')
END
GO
```

Как можно заметить, декларация форм напоминает таковую в HTML, но с гораздо более богатыми элементами стандартного графического интерфейса, не ограниченного возможностями браузера. Прикладному программисту не надо управлять соединением с сервером баз данных, внедрением SQL-запросов, обработкой сетевых ошибок и многим другим, потому что код уже находится

внутри процедур адресного пространства СУБД, обеспечивая максимальную производительность обработки данных. При этом, на минуточку, на дворе 1996 год, никто из веб-разработчиков пока не выходит за рамки *postback-ov*, динамическое обновление форм без перегрузки всей страницы, предоставляемое Ultima-S, в AJAX появится через 10 лет. А изначально поддерживаемые трёхмерные, стиля Excel, сетки-таблицы с закладками в динамических формах до сих пор доступны лишь в сторонних компонентах.

Номенклатура базовых классов и системных служб позволяла пользоваться полуфабрикатами прикладного назначения: от управления жизненным циклом документа, доступом к папкам до бухгалтерских абстракций, позволявших задействовать механизмы материального и финансового учёта. Для реализации нового типа документа и отражения его в системе управленческого учёта предприятия программисту требовалось работы на 1–3 дня.

Высокая степень изолированности ядра и тонкого клиента от прикладных модулей позволила также специализировать работу части программистов над системой, не привлекая других разработчиков, для которых более важным было понимание предметной области, чем используемых архитектур. Если за год с созданием первой версии и собственно технологии справлялось пять человек и двое дистанционно разрабатывавших тонкого клиента программистов, то в дальнейшем по мере расширения функционала штат прикладных программистов увеличился.

Однако в планах руководства было создание коробочного продукта, задача, требующая гораздо больше усилий по созданию обобщённых моделей предметных областей и функциональной архитектуры в целом. Техническая архитектура — важный, но не единственный столп разработки тиражируемых продуктов. Никакая широта технической мысли не поможет, если на платформе реализуются неадекватные по сложности задачам решения, неполные или противоречивые требования. К сожалению, эта проблема не была решена и во второй версии прикладного обеспечения системной платформы.

Приведу мнение коллеги по проекту, Владимира Иванова, аналитика и руководителя группы разработки бухгалтерской подсистемы. Роль функционального архитектора или, в терминах MSF¹, менеджера продукта он называет просто «маркетолог».

¹ Microsoft Solutions Framework — методология разработки программного обеспечения, предложенная Microsoft. MSF опирается на практический опыт Microsoft и систематизирует управление людьми и процессами софтверного строительства.

Я попробую сформулировать, что делалось неправильно и какие решения были найдены, их уже удалось применить большей частью в проектах за пределами Ultima-S на базе ее разработки.

Проектированием системы на самом деле должен заниматься не технарь, а маркетолог как Стив Джобс. Система — это товар, удовлетворяющий потребности. Маркетолог может понять потребности рынка, сформулировать требования к товару, может оценить осуществимость маркетинговой компании. На деле технический архитектор нужен маркетологу для оценки себестоимости проекта и его сроков, а также для информации о новых перспективных технологиях, тогда маркетолог сможет искать сочетания «потребность + технология». Проблема проекта Ultima-S и многих других в том, что там не было маркетолога, определяющего вид продукта и программу продвижения, что сделало бессмысленным большое количество технологических действий.

Следуйте совету Джобса: «Обычные художники заимствуют — великие воруют». Для меня, как руководителя группы разработки бухгалтерского модуля, существовала проблема: никто не мог поставить задачу. Системный архитектор был «не совсем в теме», маркетолог со знанием продукта отсутствовал. Но мне в голову пришла удачная идея начать копировать макроязык и систему настроек 1С, одновременно обеспечивая совместимость с объектной моделью Ultima-S. Если бы разработчики Ultima-S, до того как взяться за написание кода, провели тщательное изучение архитектур аналогов на рынке, все было бы иначе. Можно было бы передовую платформу Ultima-S приспособить к куче хороших идей у 1С и «Галактики». Я в то время часто общался с экспертами «Галактики», они серьезно опасались, что мы именно так и сделаем: реверс-инжиниринг и переписывание в новой технологии. Ресурсы для этого у нас были. Но, увы, сделали только частично, а 1С — это все-таки модель малого бизнеса, тогда как мы целились в сектор Enterprise, где была «Галактика».

Учитесь у лидеров рынка, их успех не от «просто так». Я получил два важных опыта в этом проекте — объектно-реляционное моделирование и проблемно-ориентированные конструкторы, которые популяризовал Нуралиев. Могу сразу сказать: Нуралиев круче. Дело в том, что его функциональная архитектура была следствием изучения потребности рынка и типовых сценариев внедрения. Наш

бухгалтерский модуль из Ultima-S до сих пор живёт и здравствует в нескольких холдингах, которые не раз оценивали миграцию на Ахарт, но оставались все равно на нём. Концепция 1C плюс возможность Transact SQL-программирования — мечта многих разработчиков бизнес-решений. Нуралиева много раз просили это сделать, но он держался за свой подход, чтобы поддержать кросс-платформенность¹.

Наследование объектов и объект-контейнер с визуальным конструктором. Ultima-S построена на идее наследования классов, как современный вариант PostgreSQL. Если брать модель 1C — это наследование только на системном уровне и отказ от наследования в бизнес-уровне в пользу конструирования путём сложения комбинаций объектов в объект-контейнер с помощью визуальных конструкторов. Модель 1C сильнее. Быстрее создаёт новые сущности, за счёт разделения системного уровня и конструктора, она также надёжнее, так как на-стройщики очумелыми ручками не могут залезть в ядро.

Прикладная разработка не место для эстетического наслаждения от красот технологий. Это бизнес. Ultima-S сильно пострадала от того, что стала чем-то вроде лаборатории Xerox в 1980-х, где масса людей апробировала разные технологии, научилась и пошла делать другие проекты в Apple и Microsoft. Архитектурные эксперименты оправданы, если маркетолог видит преимущества в архитектуре товара, которые преобразуются в удовлетворение потребностей клиентов. Надо оценивать эффективность архитектуры не в терминах красот, а в терминах трудозатрат. Пользователю все равно, он не видит что там внутри. Но он видит, что система дорабатывается медленно или валится с ошибкой. Для меня был огромный плюс, что я познакомился в этом проекте с MS Project и стал в нем анализировать трудозатраты программистов. Выводы были интересные. Я заметил, что программисты и технологии могут выглядеть невзрачно, но давать невероятные эффекты по скорости и надёжности кода в терминах низких трудозатрат, и наоборот, вроде бы красивые решения могут превращаться в «чёрную дыру» для бюджета проекта.

На деле Ultima-S показала, что идея реализации сервера приложений средствами СУБД жизнеспособна и эффективна, также она совме-

¹ Речь идёт прежде всего о переносимости между разными СУБД. Например, версия 7 работала как на файл-серверном «движке», так и с SQL Server.

стима с лучшими функциональными практиками на рынке, как, например, моделирование в 1C. Но отсутствие маркетолога на проекте в качестве его лидера не дало технологиям добиться коммерческого успеха, который пришёл уже к системам — наследникам Ultima-S.

Во многом разделяя мнение Владимира, хотелось бы добавить, что бороться за статус главного идеолога продукта контрпродуктивно в любой ситуации. Наилучшим, по моему мнению, решением является единство и противоположность технической и функциональных архитектур, развивающиеся при постоянном диалоге групп, ответственных за техническую реализацию решения и функционал продукта. Такая практика используется в рамках MSF, в том числе и в самой Microsoft.

Считать трудозатраты на разработку архитектуры и ядра достаточно сложно. Ещё труднее считать отдачу в дальней или даже средней перспективе. Если при разработке заказного проекта в первой версии приоритетом может быть именно ближняя перспектива, то в продуктовой разработке такой подход неизменно приводит к необходимости полной переделки второй версии.

С прикладной разработкой всё относительно просто: есть функционал, оцениваемый заказчиком или маркетологами в конкретную сумму, есть трудозатраты на его реализацию, остальное — прибыль от деятельности. Чтобы убедиться, насколько хороши техническая архитектура и платформа, как они влияют на производительность прикладного программирования в растущем проекте, надо более формально разделить эти два направления деятельности, чего сделано не было.

В конце концов, на любом серьёзном производстве кроме непосредственно цехов и мастерских есть лаборатории и конструкторские отделы, работающие на перспективу, а степень наукоёмкости или, как ещё говорят, высокотехнологичности продукции, напрямую зависит от доли вложений в перспективные разработки, заключённой в себестоимости товара.

Критическая масса клиентуры в проекте, выводящая его на прибыльность, не была достигнута, что послужило формальной причиной закрытия после более чем трёх лет существования. Но история на этом не кончилась.

Не прекращалась поддержка некоторых клиентов. После перерыва базовая часть системы была переименована в NEXUS и перешла в домен разработок с открытым кодом. Оказалось, что платформа является весьма эффективным средством автоматизации небольшими группами и индивидуальными консультантами с хорошим знанием технологий. Было проведено несколько новых

внедрений системы на базе только самой платформы с полной разработкой прикладного кода.

Возвращаю слово Игорю Паньшину, чей рассказ в предыдущей главе был коротким.

Я хотел бы рассказать о внедрении, на которое у меня ушло несколько лет. Думаю, всем известно, что такое оператор сотовой связи. Но мало кому известно, что такое сотовый оператор, у которого завтра отберут частоты. Таким оператором оказалась Петросвязь или РТК (Радио-Телекоммуникационная Компания), имевшая оборудование QUALCOM на частотах 800 МГц, которые решили отдать телевидению. Первыми сбежали разработчики биллинга BillOnLine, оставив работающую систему без всякой поддержки. На свою беду, как я не мог бросить тогда VAX, так не смог уйти от умирающей в прямом смысле слова системы.

Выход был найден. Бралось ядро NEXUS и компилировалось в работающую базу системы BillOnLine. Тонкий клиент сразу начинал работать. Оставалось дописать классы, которые поддерживали бы бизнес-задачи предприятия, что и было сделано. Подход имел ещё одно интересное преимущество перед всеми мне известными. Скрипт ядра позволял устанавливать его в базу данных практически всех систем MSProject, DOCSVision, 1C 7.7 и 1C 8, что сразу давало дополнительную свободу управления и способность дорабатывать штатную систему под нештатные нужды организации без привлечения разработчиков продукта. Основным преимуществом является способность решить бизнес задачи в рамках SQL-запросов и хранимых процедур на порядок или два быстрее, чем любая программная платформа.

Правда, сейчас, после 10 лет разработки и внедрения Ultima-S называется NEXUS, но это сути не меняет. В настоящий момент я работаю на перекрёстке NEXUS и 1C 8, обеспечивая нужную производительность учётной системы в целом. Это даже не разработка, а сопровождение-разработка.

Другой ключевой особенностью платформы является возможность анализа хозяйственной деятельности предприятия практически в реальном времени. Как только документ оперативного контура меняет состояние «черновика» на одно из действительных, информация отражается на счетах управленческого

учёта с заданными разрезами. Об этом подробнее рассказывает Сергей Быков, ведущий специалист ВІ.

Технические ограничения эпохи бумаги и слабых ЭВМ вынудили «зашивать» аналитику в код счета. До сих пор многие системы, спроектированные профессиональными бухгалтерами, реализуют именно этот подход, например Oracle и SAP. Получить подробные данные о деятельности предприятия из классической главной книги (ГК) невозможно: количество сегментов в коде счета ограничено очень скромным числом (4–8).

Из-за этого возникла задача анализа хозяйственной деятельности предприятия. Но по сути эта всё та же учётная задача — построение отчётов об операциях на основании заданных правил группировки типов документов. Такая группировка делается и в ГК, но с потерей существенной для принятия решений информации.

Бухгалтерский движок в NEXUS позволяет преодолеть «проклятие» учёта по плану счетов с сегментацией кода счёта. Код отражает только вид операций — кассовые, банковские, складские, производственные и т. д. Существенные признаки операций сохраняются в виде аналитических срезов. В результате получается структура, сразу пригодная для OLAP¹.

Судите сами, Saldo — таблица фактов, несколько раз соединённая с таблицей Docs (измерения). Классическая «звезда». Для того чтобы помочь OLAP-клиенту, создаём отдельные view для каждой роли таблицы Docs — клиенты, товары, счета (в понятии регистров учёта), валюты, партии и т. д.

Далее подключаем эту структуру в Excel и получаем готовый OLAP, который наполняется данными в соответствии с учётной политикой предприятия, непосредственно в момент выполнения хозяйственных операций. Это не просто анализ операционной деятельности, это сверхоперативный анализ! Такая оперативность в других, более раскрученных системах возможна только в виде ограниченных плоских отчётов, построенных по первичным документам.

¹ От англ. On Line Analytical Processing, интерактивная аналитическая обработка данных.

Я поддерживаю системы на этом «моторе» уже 10 лет — решение на удивление простое, гибкое и мощное. За это же время имел опыт использования JDEdwards OneWorld, Oracle Apps и в меньшей степени с SAP.

История системы продолжается.

Нешаблонное мышление

Моя профессиональная практика складывалась таким образом, что обсуждение обобщённых решений касалось прежде всего задач предметной области и соответствующего уровня абстракций. С некоторой натяжкой их можно отнести к аналитическим шаблонам, так как в качестве основы брались только существенные части структур, интерфейсов или даже просто рабочие идеи. Лучше назвать их аналитическими эскизами.

О типовых решениях уровня реализации речь заходила редко, но уже в середине 1990-х годов иногда возникали упоминания книги GoF — «банды четырёх»¹[6] о приёмах объектно-ориентированного проектирования, где была предпринята попытка их обобщения. Признаюсь, руки долго не доходили до непосредственного ознакомления с этим трудом, но где-то в начале годов 2000-х издание наконец попало ко мне в руки.

Имея уже немалый опыт проектировщика и программиста, я надеялся найти в книге, как минимум, более проработанный вариант приёмов, использованных в предшествующих проектах. Достаточно быстро я обнаружил, что за многолетнюю практику большинство описанных авторами шаблонов было совершенно невостребованным кругом решаемых задач. Те же оставшиеся, что удалось идентифицировать, несколько удивили простотой и не всегда отражающими суть названиями.

Полагаю, что для человека, знакомого с основами ООП, вынести зависящую от контекста операции логику в специализированные обработчики или даже в классы, имеющие единый интерфейс вызова, является несложной операцией, над которой придётся раздумывать несколько минут. Например, часто такая задачка возникает в разного рода генераторах, когда по исходной модели нужно

¹ От англ. gang of four — жаргонное название коллектива авторов первой книги по шаблонам ООП.

выдать код на том или ином языке программирования. Видимо поэтому, мне и не приходило в голову величать такую операцию «стратегией». Не ассоциировалась у меня тактическая перестановка фигур на шахматной доске со столь солидным термином.

Зачастую, ещё вчерашние новички, научившись достаточно элементарным вещам, любят порассуждать о том, что изобретение велосипедов — пустое дело. По моим представлениям на воспроизведение большинства из приводимых в книжке «велосипедов» в конкретных случаях у любого специалиста с навыками абстрактного мышления вряд ли должно уходить более часа. Это заметно быстрее изучения самой книги, ее осмысления и, наконец, осознания тех моментов, когда, собственно, надо применить абстрактный слепок в реальной жизни.

Распространённое мнение о создании некоей общей терминологии для повседневной работы программистов также не вполне подтвердилось на практике. В используемых спецификациях названиям шаблонов не находилось места. Обычно там пишется что-то вроде «реализовать документы с такими-то атрибутами, используя следующие базовые классы фреймворка» для новичка либо «реализовать функции А, Б, и В учёта договоров, используя модули X, Y и Z» для более опытного разработчика. Дело в том, что шаблоны из книги — уровня реализации и к моделированию предметной области имеют далёкое отношение. Они востребованы скорее теми, кто программирует (не хочу писать «кодирует») в группе, руководствуясь общей для всех подробной функциональной спецификацией. «Вася, здесь я сделаю адаптер над твоим классом, чтобы не напрягать тебя. — Да, Петя, спасибо». О том же, что класс имеет всего один экземпляр объекта, то есть является «синглтоном», Петя сможет догадаться по интерфейсу и без дополнительного вопроса Васе.

Факт защиты одним из авторов научной диссертации по теме книги заставил в очередной раз призадуматься о степени проникновения современной теоретической мысли в практику софтостроения. По моим представлениям, книга имела к науке весьма опосредованное отношение и являлась скорее чем-то вроде поваренной книги. Но лучше так, чем никак. Для себя же сделал вывод: если вы практикуете уже лет 5–10, то имеет смысл пролистать книжку и отложить, понадеявшись на фоновую память. Вдруг кто-то вернёт в беседе незнакомый термин — меньше времени уйдёт на выяснение, что имелось в виду. На практике же толку будет немного.

Сложнее обстоит дело с теми, кто только начинает свой путь. Прочтение сего труда новичком, как мне кажется, является прямым аналогом попадания

в прокрустово ложе диктуемой парадигмы. Потому что книга описывает набор решений, а неокрепшему за недостатком практики уму проектировщика надо научиться самому находить такие решения и пути к ним. Для чего гораздо эффективнее первое время «изобретать велосипеды», нежели сразу смотреть на готовые чужие. На чужие надо смотреть, когда придуман хотя бы один собственный, чтобы понять, насколько он несовершенен, и выяснить, каким же путём можно было бы прийти к лучшим образцам велосипедов данной модели.

Однако книга все-таки дала всплеск, и по воде пошли круги. А. Александреску в книге «Modern C++ design» начал экспериментировать с реализацией шаблонов на C++ и продвигать в массы свои велосипеды. Появились издания с достаточно абсурдными названиями. Например, «Шаблоны на C#». Постойте, коллеги, какие ещё шаблоны на C#? Оказывается, шаблоны сыграли с программистским сообществом злую шутку: они оказались ещё и зависимыми от языка программирования. То есть, прочитав «банду четырёх», писавших свой труд исходя из возможностей C++, срочно бегите за новой порцией информации, кто для Java, кто для C#. Хотя задумывались шаблоны с обобщающей практики целью.

Поясню на примере использования шаблона `Visitor`. У Александреску в самом начале описания шагов по применению шаблона на существующей иерархии классов документов, которую нужно наделить новыми функциональными возможностями (в примере шла речь о сборе статистики), в качестве отправной точки приводится следующий код:

Линейный способ

```
void DocStats::UpdateStats(DocElement& elem)
{
    if (Paragraph* p = dynamic_cast<Paragraph*>(&elem))
    {
        chars_ += p->NumChars();
        words_ += p->NumWords();
    }
    else if (dynamic_cast<RasterBitmap*>(&elem))
    {
        ++images_;
    }
    else ...
    добавляем по одному оператору if для каждого типа инспектируемого объекта
}
```

Автором совершенно справедливо отмечается существенный недостаток этого фрагмента: преобразование типов делает его сложным для сопровожде-

ния, кроме того, нет никакой гарантии, что проверка для базового класса не выполнится раньше, чем для производного. Достаточно ошибиться в порядке следования операторов `if`, и на ветку производных классов программа никогда не попадёт.

Добавлю, что если разные классы элементов документов имеют полиморфные свойства, собираемые статистикой, то задача ещё более усложняется. Например, «параграф» и «формула» могут иметь одно и то же свойство `NumChars`.

После рассуждений о недостатках линейного кода выносится решение о необходимости виртуализации вызовов путём построения иерархии, аналогичной существующей иерархии классов, и добавления новых методов в неё. То есть реализации шаблона `Visitor`, описание которого на добрых двух десятках (!) страниц вы можете посмотреть в книжке. Если очень захотите.

Теперь представьте, что вы используете другой язык с развитым механизмом интроспекции типов времени выполнения. Например, отражение (*reflection*) для .NET. В этом случае «лобовое» решение может выглядеть примерно так:

Использование отражения

```
class DocStats
{
...
    void UpdateStats(DocElement elem)
    {
        Type elemType = typeof(DocElement);
        BindingFlags flags = BindingFlags.Public | BindingFlags.NonPublic |
BindingFlags.Instance;

        if (elemType.GetMethod("NumChars") != null)
            chars += (int) elemType.InvokeMember("NumChars", flags | BindingFlags.
InvokeMethod,
            null, elem, null);
        if (elemType.GetMethod("NumWords") != null)
            words += (int) elemType.InvokeMember("NumWords", flags | BindingFlags.
InvokeMethod,
            null, elem, null);
        if (elemType.GetProperty("Image") != null)
            images++;
        ...
        обследуем все интересующие нас свойства классов
    }
}
```

Проблем с преобразованием типов и порядком вызова методов нет, поэтому никаких усложнений с виртуализацией, использующихся в шаблоне для C++, не требуется.

Более того, не требуется и сам шаблон!

Можно пойти и другим путём несложной и безопасной реструктуризации: извлечь существующие в неявном виде интерфейсы элементов документа в явные определения и обследовать в методе сбора статистики передаваемый объект на предмет реализации классами интерфейсов. Например, `if (elemType is IParagraph), is IImage, is IFormula` и т. д. Тогда можно обойтись вообще без отражения.

Наконец, существует и другое специфичное для C++ решение: реализовать аналогичную иерархию с заданной виртуальной операцией и, далее, используя множественное наследование, «подмешать» эти реализации к существующей иерархии, используя в дальнейшем только полученные классы-миксты.

Весьма элегантное решение на основе *class helpers* есть в Delphi. Мы просто дописываем недостающие методы к существующим классам, причём их исходники для этого не требуются.

Итого на простом примере имеем целый букет решений вместо одного шаблонного, из которых можно выбирать оптимальный. Попытавшись однажды объяснить подобное программисту, видимо, несколько увлечёшемуся шаблонами, я не встретил понимания.

Несколько позднее я наткнулся в магазине на книгу с названием, претендующим на звание наиболее абсурдного из встречавшихся. Оно звучало как «Thinking in patterns». В переводе на русский язык — «Мыслить шаблонно». Ещё недавно привычка шаблонно мыслить считалась в инженерном сообществе признанием ограниченности специалиста, наиболее пригодного для решения типовых задач с 9 утра до 6 вечера. Теперь не стесняются писать целые книжки о том, как научиться шаблонному мышлению...

Думать головой

Серия коротких заметок была задумана, как некий противовес механистическому подходу к программированию, пропагандируемому различными учебниками шаблонов. Потому что думать надо не шаблонами, а головой. Начнём с того, что кажется очевидным.

Обобщение

Откажитесь от термина «наследование», который искажает смысл действий. Мы обобщаем. Обобщение (наследование) реализации или интерфейсов — весьма

неоднозначный механизм в объектно-ориентированном программировании, его применение требует осторожности и обоснования.

ОСНОВНОЕ ПРАВИЛО

Обобщаемые классы должны иметь сходную основную функциональность.

Например, если два или более класса:

- порождают объекты, реализующие близкие интерфейсы;
- занимаются различающейся обработкой одних и тех же входных данных;
- предоставляют единый интерфейс доступа к другим данным, операциям или к сервису.

Взгляните, три примера, и уже несколько шаблонов оказались ненужными: (1) — «фабричный метод» (*factory method*), (2) — «стратегия» (*strategy*), «шаблон метода» (*template method*) и (3) — «адаптер» (*adapter*). В принципе, можно для случая (1) ещё и «прототип» (*prototype*) записать: если в среде нет развитой поддержки метайнформации типа отражения (*reflection*), то реализовать клонирование, скорее всего, придётся в потомках общего предка. Далее, в большинстве случаев (если не во всех) вместо «посетитель» (*visitor*) можно использовать все тот же «шаблонный метод» или вызов метода через *reflection*, как уже было описано в предыдущей главе. Ещё один исключаем.

Почему шаблоны «вдруг» стали ненужными? Потому что требуется только обобщение, причём в перечисленных случаях необходимость операции более чем очевидна. Требования же исходят напрямую из вашей задачи. Корректно проведя обобщение вы автоматически получите код и структуру, близкую к той, что вам предлагают зазубрить и воспроизводить авторы разнообразных учебников шаблонов.

Одно понятие и три очень часто встречающихся на практике случая гораздо эффективнее для запоминания и использования, нежели абстрактные картинки и текст шести шаблонов.

Основная ошибка — обобщение по неосновному функциональному признаку. Например, когда обобщают сотрудников и пользователей. Или заказы на по-

ставку и накладные на отгрузку. Или яблоки с теннисными мячиками. Почему нет, они же разноцветные, круглые и продаются в магазинах?

Ошибка приводит к построению иерархии по неосновному признаку. Когда внезапно найдётся ещё один такой признак, возможно, более существенный с точки зрения прикладной задачи, обобщить больше не удастся: придётся ломать иерархию или делать заплату в виде множественного наследования, недоступного во многих объектно-ориентированных языках. Или пользоваться агрегацией.

Не увлекайтесь обобщением. Ошибки тоже обобщаются и уже в прямом смысле этого слова наследуются. Исправление по новому требованию может привести к необходимости сноса старой иерархии, содержащей ошибки.

ВОЗЬМИТЕ ЗА ЭМПИРИЧЕСКОЕ ПРАВИЛО

Глубина более двух уровней при моделировании объектов предметной области, вероятнее всего, свидетельствует об ошибках проектирования.

Для построения устойчивой глобальной иерархии необходим серьёзный анализ предметной области, ведь не случайно создание таксономии — сложная научно-исследовательская работа, которой в крупных компаниях занимаются аналитики. Но и такой работы будет недостаточно, если, например, предполагается использование модуля (библиотеки, компонента, службы) в нескольких смежных областях. Класс «Книга» для библиотеки, магазина и читателя — это три разных взгляда на одну и ту же сущность с отличающимися ассоциациями и обобщениями. Ещё сложнее дело обстоит с классом «человек». Поэтому не спешите наследовать «менеджера» и «охранника» от класса «сотрудник» вне рамок учёта кадров, ведь они ещё и материально-ответственные лица, руководители или участники проектов, контактные лица, граждане, родители своих чад, налогоплательщики, собственники, вкладчики, заёмщики, автомобилисты...

Про сборку мусора и агрегацию

Достаточно широко известна проблема принадлежности объектов как друг другу, с образованием соответствующей иерархии, так и графу вызовов функ-

ций (подпрограмм). По словам М. Донского [11], наличие в некоторых языках механизма сборки мусора, является примером отказа от самой идеи справиться с этими проблемами и молчаливым признанием возможности присутствия в среде объектов, не принадлежащих ни подпрограммам, ни другим объектам.

Итак, сборщик мусора, он же GC — *garbage collector* в средах программирования с автоматическим управлением памятью. Наиболее очевидное преимущество — программисту не надо заботиться об освобождении памяти. Хотя при этом все равно нужно думать об освобождении других ресурсов, но сборщик опускает планку требуемой квалификации и тем самым повышает массовость использования среды. Но за все приходится платить. С практической стороны недостатки сборщика известны, на эту тему сломано много копий и написано статей, поэтому останавливаться на них я не буду. В ряде случаев недостатки являются преимуществами, в других — наоборот. Черно-белых оценок здесь нет. В конце концов, выбор может лежать и в области психологии: например, я не люблю, когда компьютер пытается управлять, не оставляя разработчику достаточных средств влияния на ход процесса.

Рассмотрим типовой пример, когда сборщик мусора спасает от ошибки программирования, но не спасает от ошибки проектирования. Речь о контейнере, являющемся владельцем своих объектов. Наиболее распространённой ошибкой является сохранение ссылок на эти объекты в другом объекте вне контейнера. При этом часто оказывается, что ссылки ещё живы, но указывают в пустоту, потому что контейнер уже удалён. В случае «ручного» управления в традиционных языках, таких как C++, при обращении по ссылке возникнет ошибка, ведущая к сбою или отказу. При наличии сборщика мусора программа продолжит работу, хотя объекты так и останутся висеть в памяти. Конечно, приятно осознавать, что программа не свалится с ошибкой, а продолжит работу. Особенно если это относительно критичное серверное приложение. Но проблема-то остаётся. Например, «висящие» объекты могут продолжать использовать или даже блокировать системные ресурсы. А могут и просто занимать недопустимо много памяти.

Решение здесь достаточно простое.

НУЖНО ВЗЯТЬ ЗА ПРАВИЛО, ЧТО

контейнер всегда управляет своими объектами. Поэтому обращаться к его внутренним объектам нужно только через интерфейс самого контейнера.

При этом быть готовым к обработке ситуации, когда контейнер говорит: «Извини, но такой объект уже удалён или пока недоступен». Если же объект переходит во владение к другому контейнеру, то он перестаёт управляться прежним. И процесс передачи объекта и управления также не может быть реализован простым присваиванием полученной ссылки.

Метафора из жизни. Вам нужна цитата из книги, библиотечный код которой вы знаете. Пусть цитата занимает одну страницу в книге, её номер вы знаете. Библиотека — контейнер книг. Книжный магазин — тоже. Варианты взаимодействия:

1. Пойти в библиотеку и взять книгу на время. У контейнера остаётся ссылка на вас, если потом книгу будут снимать с учёта (удалять), то о вас вспомнят и попросят вернуть.

```
//Правильно:  
читатель.Книги.Добавить(библиотека.Выдать(код, читатель));  
//Ошибка:  
читатель.Книги.Добавить(библиотека.Книги(код));
```

Кстати, ошибка в данном примере означает, что книгу, между нами говоря, вы попросту спёрли, а интерфейс библиотеки позволяет это легко сделать.

2. Обратиться в справочную службу библиотеки и попросить их прислать копию нужной страницы.

```
//Правильно:  
читатель.Реферат.Цитаты.Добавить(библиотека.КопироватьСтраницу(код,  
номер_страницы));  
//Ошибка:  
читатель.Реферат.Цитаты.Добавить(библиотека.Книги(код).Страницы(номер_  
страницы));
```

3. В библиотеке книги не оказалось, купить книгу в магазине.

```
//Правильно:  
покупатель.Книги.Добавить(магазин.Продать(код, покупатель));  
//Ошибка:  
покупатель.Книги.Добавить(магазин.Книги(код));
```

Ошибка в данном примере снова означает, что книгу вы украли. Два совпадения. Случайные ли?

В случае явного управления памятью возникнет системная ошибка: библиотека или магазин полезут без спросу в ваше отсутствие в квартиру за объектом и заберут его назад. Придя домой, вы не обнаружите на полке нужной книги.

В случае же со сборщиком мусора факт кражи книги не вызовет внешних проблем. В самом простом случае библиотека просто удалит ссылку на пропавшую книгу из своих каталогов. Сама книга при этом продолжит занимать место на вашей полке. Но может случиться, например, без вашего ведома из книги будут удалены подцензурные страницы, ведь ссылка-то известна.

Оба случая являются логической ошибкой проектирования интерфейсов, но сборщик мусора на некоторое время скроет их от вас. В этой ипостаси сборка мусора, позволяющая программистам избавиться от назойливых ошибок обращения к памяти (*access violation*), является лишь фиговым листочком, прикрывающим до некоторого момента всё те же старые проблемы. Искренне всем желаю, чтобы этот момент не проявился во время приёмки.

Журнал хозяйственных операций

Отвлечёмся ненадолго от общих концепций и рассмотрим более конкретные проблемы разработки учётных приложений. Вопросы, подобные «почему нельзя хранить остатки в форме текущих величин», «зачем нужна история операций», «зачем там нужна транзакция в режиме «сериализация» (что такое режим *serialized* можно прочитать в статье [20]), не раз всплывали в дискуссиях, поэтому я кратко расскажу об этом в рамках отдельной главы, чтобы в следующий раз просто ссылаться.

Довольно сжатое изложение статьи «Как проектировать бухгалтерию» [19] в терминах абстракций может быть не очень понятным начинающим. Напротив, статья «Введение в складской учёт» [18] рискует показаться излишне упрощённой и «заточенной» на складскую бухгалтерию с отраслевой спецификой. И в обоих случаях не хватает конкретики, разъясняющей вышеназванные вопросы.

Начнём с первого анти-шаблона «Таблица остатков», на которые я вдоволь насмотрелся во времена буйного расцвета торгово-складского софтверостроения в 1990-х годах. Начинающий проектировщик рассуждает так: мне нужно текущее количество товара в наличии, а все движения, в результате которых эта цифра и появилась, можно оставить в стороне, а то и прямо в первичных документах.

Представьте, что в документ недельной давности вкралась ошибка. Её исправили, причём пересчитанные текущие остатки по-прежнему неотрицательны. Значит ли, что они неотрицательны и на каждый день прошедшей недели? Разумеется, нет. Приходит клиент и говорит: «Мне выписали 10 штук, а на складе только 8, я на вас, жуликов, в суд подам». Парадокс? Никакого парадокса. За товаром он пришёл сегодня, но продали ему товар вчера. А на состояние «вчера» после корректировки остаток был бы отрицательным. Вот ему и не хватило.

Если не верите, что такое возможно, вот схема движения.

Дата	Закупка / продажа	Товар	Количество	Клиент
2012-04-01	Закупка	Утюг	12	ООО «Поставщик»
2012-04-08	Продажа	Утюг	10	Клиент «Недовольный»
2012-04-08	Продажа	Утюг	2	Клиент «Проворный»
2012-04-09	Закупка	Утюг	2	ООО «Поставщик»
Остаток на сегодня 2012-04-09		Утюг	2	

Теперь откорректируем приход 2012-04-01 с 12 утюгов на 10. Получаем, что на сегодня их 0. Вроде бы все в порядке. Всё, да не совсем: сегодняшняя закупка ещё не поступила на реализацию. Поэтому вчерашний «минус 2» пока действителен.

Почесав свой мыслительный орган, проектировщик приходит к неутешительному выводу: даже для фактических операций нужно считать остаток по истории (журналу). Не говоря уже о резервировании товара, где ситуация меняется гораздо быстрее: то тут отменили, то там подтвердили.

Но считать по журналу:

- может быть долго;
- необходимо защитить считанные значения, чтобы при последующей записи не возникло «минусов».

Последний пункт требует пояснений. По сути, это и есть та самая сериализованная транзакция. Она гарантирует, что считанные значения не будут изменены другой транзакцией. То есть продажа не будет давать отрицательный остаток, если между операцией расчёта остатка и расхода вклинится другой. Это просто, смотрите.

Момент	Продавец 1	Продавец 2
1	Расчёт остатка (10 утюгов)	
2	Анализ остатка ($10 - 5 = 5$) — можно продавать	Расчёт остатка (10 утюгов)
3	Расходуем 5	Анализ остатка ($10 - 7 = 3$) — можно продавать
4		Расходуем 7

В итоге молодцы-продавцы, нечаянно воспользовавшись ошибкой программиста, спланировали клиентам 12 утюгов, хотя в наличии было 10.

Если же ваши операции расчёта, проверки и расхода работают в одной транзакции на уровне изоляции «сериализация», то такая ситуация исключена. «Продавец 2» из примера будет ждать, пока «Продавец 1» закончит операцию и в свою очередь убедится, что утюгов уже не 10, как было на экране в момент заказа, а только 5.

Момент	Продавец 1	Продавец 2
1	Расчёт остатка (10 утюгов)	
2	Анализ остатка ($10 - 5 = 5$) — можно продавать	Ожидает
3	Расходуем 5	Ожидает
4		Расчёт остатка (5 утюгов)
5		Анализ остатка ($5 - 7 = -2$) — нельзя продавать
6		Операция не прошла

Почему нельзя просто заблокировать всю таблицу-журнал, а надо использовать какие-то хитроумные транзакции с непонятным уровнем изоляции?

Объяснить это тоже просто. Представьте, что один продаёт утюги, а второй — гладильные доски. Если заблокировать таблицу, то второй продавец всё равно будет ждать первого. Всегда. И вообще, все и всегда будут ждать одного, подобно очереди в общественный туалет с одной кабинкой. Кстати, именно эта метафора наиболее употребительна при объяснении работы механизма бинарного семафора — *мьютекса*¹.

Здесь я должен отметить, что использование транзакции в режиме *serialized* вполне может привести на физическом уровне к блокировке всей таблицы. Но это уже проблема следующего порядка, которую вы будете решать самостоятельно или в содружестве со специалистом по СУБД.

Рассуждая, мы плавно подошли к вопросу «Зачем хранить историю остатков?». Действительно, ведь есть журнал, всё можно посчитать по состоянию на любой период.

Вернёмся к примеру корректировки документа недельной давности. Таблицы остатков у нас уже нет, а мы должны рассчитать и проверить на «минус» все дни за последнюю неделю. Допустим, ваша программа правильная, использует соответствующие транзакции и считает остаток от операций одного дня примерно за 500 миллисекунд. Умножим на 7, получим, что с большой вероятностью примерно 3,5 секунды пользователи учётной системы будут ожидать окончания вашей операции.

Это, мягко говоря, нехорошо. К тому же, если ваш расчёт не написан на языке СУБД — SQL, то цифра в 500 миллисекунд явно завышена в разы.

Придётся нам возвращаться к «таблице» остатков. Но совсем не к такой таблице, что была вначале, а к новой, называемой «сальдо» или «итогами». Мы добавим туда ещё один важный разрез — период. И будем поддерживать остаток на заданный период в актуальном состоянии. Тогда ваша программа просто должна попытаться отменить операцию и посмотреть, нет ли в сальдо «минусов», начиная с даты аннулированного документа. Здесь тоже возможна блокировка, но, во-первых, менее вероятная, а во-вторых, более быстрая, так как мы просто производим вычитание количества от уже рассчитанных остатков, начиная с даты вместо полного пересчёта по журналу операций.

Как поддерживать актуальность остатков? Скорее всего, триггером (кто сказал «материализованные виды?»). Это будет весьма критичный код в вашей систе-

¹ От англ. mutex, mutual exclusion — «взаимное исключение», одноместный семафор, служащий в программировании для синхронизации потоков.

ме, поэтому придётся, не побоюсь этого слова, «вылизывать» его так, чтобы он выстреливал за единицы миллисекунд. Дам совет начинающим: учите сиквел, транзакции, уровни изоляции, и будет ваша система быстрой и надёжной. И не только в примере с учётной системой, но и вообще по жизни. Опытным же разработчикам есть поле для оптимизации и дальнейшего совершенствования решений, включая альтернативные подходы.

UML и птолемеевские системы

Для того чтобы было легче подступиться к теме унифицированного языка моделирования UML, уже упоминавшегося в предыдущих главах, необходимо сделать небольшое историческое отступление [21].

Что же заставило учёных отказаться от системы Птолемея? Ответ с позиций современной теории познания мы видим в следующем. Система Птолемея описывала движение планет, видимое с Земли, то есть описывала явление. Если же мы оказались, например, на Меркурии или Марсе, то земную птолемеевскую систему нам пришлось бы упразднить и заменить новой.

Система Коперника сумела схватить сущность взаимного движения планет Солнечной системы. Такое описание, говоря современным языком, уже не зависело от того, какую планету в качестве системы отсчёта захочет выбрать себе наблюдатель.

С точки зрения теории познания объективной истины теологи совершали грубейшую ошибку: они сущность подменяли явлением. Наблюдаемое с Земли движение планет по небосводу они считали их действительным движением относительно Земли...

В UML должно настораживать уже самое первое слово — «унифицированный». Не универсальный, то есть пригодный в большинстве случаев, а именно унифицированный, объединённый. Силами небольшого количества экспертов, если точнее, сначала двух, позже к ним присоединился третий, был создан сборник, содержащий наиболее удачные нотации из их собственной практики. Сборник стали продвигать в массы, а за неимением лучшего, и в стандарты.

Стандарт получился несколько странным. В его названии присутствует слово «моделирование». Модель от картинки в графическом редакторе отличается наличием базового формализма, позволяющего проверить созданную конструкцию на непротиворечивость, а если повезёт, то и на полноту. Очень простой, но верный признак проблемы — отсутствие в инструменте моделирования команды «Проверить». И если условной кнопки «Check model» в панели не предусмотрено, то ваш CASE¹ является просто продвинутым редактором специализированной векторной графики с шаблонами генерации кода.

Для сравнения, в IDEF0² проверка целостности имеется, что не даст аналитику нарисовать в своей модели возникающие из пустоты и исчезающие в никуда объекты. Концептуальная модель данных в любой нотации, ER³ или IDEF1x, также обратит внимание проектировщика на «висячие» сущности, циклические связи и прочие моменты, скорее всего, являющиеся ошибкой. На уровне логической модели данных (реляционная) к вышеперечисленным проверкам присоединятся новые, например, связанные с ограничениями типов (доменов), наличия ключей и другие. UML работает на логическом уровне.

Пишу я об этом вовсе не потому, что IDEF-стандарты такие крутые, несмотря на то, что системные аналитики ими часто пользуются. Просто небольшой сравнительный пример из закровов собственной практики.

Наконец, главное слово, «язык». Графические образы — это, конечно, аналог слов, но не всякий набор слов является языком. Поэтому UML до языка ещё далеко. Язык, даже естественный, можно автоматически проверить на формальную правильность. Входящий в UML OCL хоть и является языком, но выполняет лишь малую часть работы по проверке, и не собственно модели, а её элементов. А о способностях проверить модель написано выше.

Всё сказанное не должно вас смущать и поражать новизной. Ведь и XML тоже не язык, несмотря на соответствующую букву в аббревиатуре, а технология создания языков пользователя на базе разметки и формальных правил её проверки посредством DTD или схем.

Итак, UML:

¹ От англ. Computer-Aided Software Engineering — набор инструментов и методов для проектирования программного обеспечения.

² От англ. ICAM Definition (Integrated Computer-Aided Manufacturing) — семейство методологий для моделирования систем.

³ От англ. entity-relationship model — модель данных, позволяющая описывать концептуальные схемы предметной области.

- не универсальный, а унифицированный, объединяющий отдельные практики;
- не язык, а набор нотаций (графических);
- не моделирования, а в основном рисования иллюстраций, поясняющих текст многочисленных комментариев.

Поэтому использование UML имеет две основные альтернативы, напрямую зависящие от целей:

- **Цель — использовать «как есть».** Не заниматься вопросами целостности, ограничившись рисованием частей системы в разных ракурсах. Если повезёт, то часть кода можно будет генерировать из схем.
- **Цель — использовать для моделирования и генерации кода.** Придётся создать свои формализмы, соответствующие моделируемой области. В самом минимальном варианте — использовать в принудительном порядке стереотипы и написанные руками скрипты и ограничения для проверки непротиворечивости такого использования. Чтобы, например, стереотип «Водитель» в рамках ассоциаций «Управление машинами» был связан только с классом, реализующим интерфейс «Транспортное средство».

Во втором случае формализация модели является, по сути, созданием предметно-ориентированного языка, то есть серьёзной исследовательской работой вплоть до уровня научной диссертации. Далеко не каждая софтверная фирма может позволить себе вести такие работы без ясных перспектив тиражирования своих продуктов.

Поэтому в большинстве ситуаций программисты находятся в «случае номер один», да и то не всегда. Потому что вместо «порисовали картинки и пиши код» все может ограничиться «пиши код и не отвлекайся на теории яйцеголовых».

С картинками в UML сложилась некоторая неразбериха вкупе с излишествами. Например, диаграммы деятельности (*activity diagram*), состояний (*state machine diagram*) и последовательностей (*sequence diagram*), а также их многочисленные подвиды по большому счёту описывают одно и то же — поток управления в программе.

У Киплинга есть замечательный рассказ «Как было написано первое письмо». Про неудачный опыт использования графической нотации первобытными

людьми. Но если в рассказе все закончилось хорошо, то в современном проекте была ситуация, когда двум проектировщикам по отдельности поручили делать диаграммы переходов для одного и того же набора классов. Получилось очень «унифицированно»: найти хотя бы одну пару общих элементов в двух диаграммах было затруднительно. Если методология проектирования действительно существует, то работа двух людей даёт сопоставимый результат.

С другой стороны, для фиксации знаний программиста о предметной области основой являются прецеденты, варианты использования (*use case diagram*) из эллипсов, «палка, палка, огуречик» — человечков и комментариев. В UML рекомендуется использовать комментарии для отражения требований к системе, а сами прецеденты неформальны. При отсутствии проработанной системы стереотипов приходится, например, отделять приходные документы от расходных с помощью раскраски. Прочитую фрагмент из главы 8 адаптированного перевода «Введения в RUP» [22].

...Диаграмма прецедентов показывает деловые субъекты, деловые прецеденты, пакеты деловых прецедентов и связи между ними. Никаких строгих правил относительно того, что нужно показывать в диаграммах прецедентов, не существует. Вы показываете те связи в модели, которые считаете интересными...

Столь длинное вступление к главе напрямую связано с так называемым анализом прецедентов в UML. Это идеальный инструмент для создания птолемеевых систем. Только если модель Птолемея является геоцентрической, то прецеденты использования — модель заказчика-центрическая.

Прецеденты, как и геоцентрическая модель, могут удовлетворительно описывать явление, не касаясь его сущности. Всякий раз разработчики очередного корпоративного приложения создают свою «птолемеевскую систему», которая при попытке примерить ее на другую фигуру внезапно оказывается неподходящей и нуждается в серьёзной перекройке. Так и земная птолемеевская система не работает для наблюдателя, находящегося на Марсе. Если же прецеденты рисуются коллективом, то будет уместно также вспомнить притчу про слона и семь слепых мудрецов, так и не пришедших к единогласию о том, что же они на самом деле ощупывают.

Явление зависит от условий наблюдения. Сущность от этих условий не зависит.

Применительно к разработке программных систем. Вы описываете какой-то отдельный прецедент в том виде, в котором наблюдаете. Или, скорее всего, со слов участников. Например, «оплата в кассу». Пока расчёты наличные, вы наблюдаете перемещение монет и купюр из кошелька покупателя в кассу продавца. И считаете, что движение денег — это физическое перемещение монет. В один прекрасный день возникнет необходимость принимать к оплате безналичные средства, например карты. Возникают проблемы с прежней постановкой: ничего физически не перемещается. Имеем уже два прецедента: «оплата в кассу наличными» и «оплата в кассу безналичными». Послезавтра возникнет новый способ оплаты — в кредит, но операцию покупки проводят все в той же кассе. Потом с учётом скидок. Потом с учётом истории покупок по «клубной» карте и так далее.

Подобно уточняющим геоцентрическую модель эпициклом, на диаграмму накручиваются всё новые прецеденты. Видимо, поэтому в RUP рекомендуют не увлекаться наворачиванием прецедентов друг на друга (*avoiding functional design*). Исходя из этого совета можно легко перейти в иную крайность, увидев «альтернативу» в том, чтобы побольше писать кода для заказчика и поменьше строить моделей. Хотя сущность явления достаточно проста: независимо от вида расчёта, скидок, кредитов и прочего во всех прецедентах имеет место движение денежных средств, суть взаимных обязательств, выраженных в общепринятом количественном (денежном) виде. В большинстве случаев оптимальным средством для моделирования этого процесса является регистрация количественного выражения обязательств в системе счетов предприятия.

Альтернатив, к сожалению, немного. Об основной — создании собственных предметно-ориентированных языков, мета-языков — уже сказано, и не раз, в том числе в рамках описаний конкретных систем. Далеко не факт, что для этого будет нужен UML хоть в каком-то виде.

Отрасль же в очередной раз оказалась в интересном положении, когда «лучше такой UML, чем никакой». Слишком часто формулировка «лучше плохой, чем никакой» стала широко применяться, что настораживает. Но прогресс, развивающийся по законам бизнеса, неотвратим, даже если слово «прогресс», при наличии на то оснований, заключать в кавычки.

Из положительных моментов UML и поддерживающих его сред необходимо отметить, что в ситуации, когда задача складывается, как мозаика, из обрывочных сведений представителей заказчика, прецеденты помогут хоть как-то формализовать и зафиксировать неиссякаемый поток противоречивых требований. Неважно, что мозаика может, скорее всего, не сложиться в стройную картинку,

но при отсутствии экспертизы в предметной области риски недопонимания и недооценки снижаются. Наиболее проработанная часть UML — диаграмма классов — при определённых усилиях и дисциплине программистов, конечно, поможет вам автоматизировать генерацию части рутинного кода приложения.

Большой интерес представляют так называемые «двусторонние» инструменты (*two-way tools*), тесно интегрированные со средами программирования и позволяющие непосредственно во время разработки оперировать диаграммами с одновременным отражением изменений в коде и, наоборот, импортировать меняющийся руками код в модель. Таковы, например, ModelMaker или Together. Но если для повышения продуктивности отдельного программиста такой подход даёт хорошие результаты, то необходимость синхронизации моделей при коллективной работе над общим кодом может свести преимущества к минимуму.

Для тиражируемых продуктов и ориентированного на специализации проектного софтверостроения наиболее интересен подход с разработкой собственных языков, где необходимость использования UML неочевидна. Остаются, по большому счёту, заказные проекты без видимых перспектив повторных внедрений, где само слово «моделирование» может вызвать реакцию «пиши код, мы тут проектируем только снизу-вверх».

Когда старая школа молода

Изменения происходят столь быстро, что приходится записывать в ряды старой школы системы с приложениями на последних версиях автономного Visual Basic 6, безвозвратно растворившегося в бурлящей пучине .NET. Теперь, спустя немногим более десяти лет, когда начинают всплывать проблемы с поддержкой из-за отсутствия специалистов, принимаются решения о переделке систем в новой технологии с минимальными изменениями функциональной полезности или вовсе без оной. Тут-то и могут вскрыться интересные подробности.

В одной крупной электрической компании имелась база данных, собирающая эксплуатационную и прочую информацию за несколько лет от разнообразных датчиков и устройств в распределительной сети национального масштаба. Некоторые параметры отличались очень коротким интервалом замеров — «каждые N секунд». В итоге размер хранилища составляет около 3 терабайт информации за несколько фиксированных лет эксплуатации.

В общем, такой размер не является маленьким и для промышленных СУБД, требуя дополнительных усилий в проектировании, реализации, настройке и поддержке. Но пикантность ситуации состояла в том, что база данных была организована в виде двоичных файлов собственного формата.

Признаюсь, последний раз я видел такой подход в далёком 1994 году, когда мы переводили паспортные столы Санкт-Петербурга с подобных плоских файлов на прогрессивный по тем временам и, что важнее, открытый формат DBF под Clipper и FoxPro для MS-DOS и Novell-серверов. С оговорками перехода от *флоппи-нет*¹ на каналы удалённой связи, многие паспортные службы долгое время продолжали в этой системе работать, характерные алфавитно-цифровые экраны в DOS-окошке я наблюдал уже в начале 2000-х годов.

В долговременной памяти всплыли воспоминания случаев необходимости ручного декодирования форматов. К великому счастью, программисты системы не занимались тогда «эволюционной» разработкой, а знали область, в которой работают, и добросовестно поддерживали проектную документацию в актуальном состоянии. Кроме возможности прочесть собственно об устройстве системы, все описания форматов двоичных файлов также были на месте. Возникший было тёмный призрак индустриальной археологии растворился в лучах восшедшего солнца.

Поскольку в нашу задачу входило предпроектное обследование, начались интересные опыты с технологиями новыми. Начали с прототипа конвертера, извлекающего данные из старых форматов и распределяющего их по таблицам промышленной СУБД, в качестве которой выступал SQL Server. Логично было бы написать такой прототип на C++ или другом языке с возможностями прямого доступа к памяти на уровне битов, но ввиду планируемой общей разработки исключительно на C# необходимо было оценить соответствующую программу в ипостаси распаковщика байтов.

Быстро выяснилось, что побитовые операции с типами данных разной длины не являются сильным местом языка и платформы в целом. Как, например, задать `ushort` или вообще любую константу в двоичном виде? Никак. А `ushort` в шестнадцатеричном? Тоже никак, константа определяется как `int`, все остальные преобразования надо делать в небезопасном контексте `unchecked()`. В итоге код достаточно простого алгоритма пестрел вот такими вставками, которые, на

¹ От англ. *floppy net* — жаргонное обозначения псевдосети передачи информации между устройствами с помощью гибких дисков. В современном варианте звучало бы как «флэшка-нет».

минуточку, суть рекомендуемые практики. Нас настигла расплата за автоматическое управление памятью.

```
Convert.ToUInt32("00110000", 2);  
unchecked((ushort) 0xC0);
```

Вторым пунктом была собственно база данных. На логическом уровне структура для тестируемого случая никаких сложностей не представляла: одна узкая длинная таблица, связанная с несколькими короткими справочниками, практически «звезда», но в режиме постоянного добавления пачек новых записей. Трудность представлял собой уровень физический. Втиснуть число-признак, кодируемое тремя битами, в те же три бита на уровне СУБД стандартные средства не позволят. Минимальный размер — байт. Использовать нестандартные битовые поля в принципе можно, но тогда мы лишимся поддержки ссылочной и прочей целостности, возможности использовать SQL для выборки без перупаковки значений в битовые структуры и фактически будем использовать СУБД в качестве продвинутого аналога файлового хранилища с улучшенными функциями администрирования.

Поскольку целью было упрощение работы с пользовательскими запросами, то лишать их доступа к данным посредством прямого и понятного человеку SQL было решительно невозможно. При использовании минимальных по размеру стандартных типов данных прогнозируемый размер базы данных без учёта дополнительных индексов превысил 6 терабайт, то есть минимум удвоился по сравнению с двоичными файлами. Сжатие данных несколько скрасило ситуацию, полученные 4,5 терабайта выглядели уже неплохо. За всё надо платить, в том числе за стандартность доступа и представления данных.

Третьим пунктом была загрузка данных из C#-приложения в базу данных. Поскольку интенсивность вставки из прибывающих от датчиков текстовых файлов составляла почти 400 тысяч записей каждые 10 минут, то, разумеется, приложение должно было справляться с этой задачей за меньшее время. Оптимальной, как и следовало ожидать, оказалась пакетная загрузка, игнорирующая ограничения целостности РСУБД. Такова обычная цена компромисса.

Что имеем по итогам более чем 10-летнего развития технологий? Появилась возможность стандартизировать хранение и доступ к большому массиву данных, используя вполне обычное серверное оборудование корпоративного класса и 64-разрядную промышленную СУБД. Возникла необходимость перестраивать программное обеспечение из-за утраты поддержки среды разработки поставщиком и соответствующих компетенций на рынке труда.

Но я совсем не уверен, что ещё через 10 лет новые подрядчики, вынужденные в очередной раз переписывать систему, найдут документацию в том же полном виде, в котором нашли её мы. Если вообще найдут хоть что-то, кроме кода, остатков презентаций и наших отчётов.

«Оптисток», или распределённый анализ данных

Название «Оптисток» придумал уже позже кто-то из маркетинговой команды. Поль, руководитель проекта и ответственный за функционал, был первое время недоволен, так как предыдущая версия продукта называлась «Оскар». Поэтому корневой директорий системы так и живёт до сих пор под названием Oscar2.

Организация процесса получилась близкой к подходу MSF, но с учётом относительно небольшого масштаба. Из рабочих групп оставалось, с одной стороны, только управление продуктом, совмещённое с управлением проектом, где кроме Поля были задействованы ключевые пользователи и представитель директрата. С другой — была объединённая группа архитектуры и разработки за моей ответственностью и основным участием вкупе с координацией привлекаемых к проекту специалистов по SAP, корпоративному хранилищу данных и графике. Спираль процессов потребовала всего полторы фазы с одним прототипом.

С оговорками, я работаю в такой организации со студенческих времён и считаю её наиболее эффективной и комфортной для разработки продукта или заказного проекта. Опыт Microsoft показывает хорошие возможности масштабирования процесса. Но мы ещё вернёмся к этому вопросу.

Если применить к «Оптистоку» действующие классификации, то это система класса SFA (Sales Forces Automation — автоматизация продаж на выезде), предназначенная для выстраивания каналов сбыта, ориентированных на конкретного клиента. Рынок — автомобильные компоненты, запчасти и тому подобное. С другой стороны, система занимается аналитической обработкой данных и относится к BI. Типовое использование проходит по следующему сценарию.

Приходит торговый агент компании к клиенту со своим ноутбуком, запускает программу, выбирает номенклатуру, параметры поиска и ограничения, задаёт целевые ориентиры, например, увеличение продаж по данной продуктовой линии на 5 % при складской политике «запас на 7 недель», выбирает географию продаж клиента и запускает расчёт. В итоге получается прогноз-анализ на основе оценки:

- продаж компании в выбранной географии. Например, данные отгрузки по стране клиента;
- парка автомобилей выбранного сегмента рынка, также с учётом географии. Например, вся Голландия или только 75-й департамент Франции.

После чего клиенту предлагается пополнить склад некоторым количеством товаров на базе этих расчётов.

Архитектура подобных систем, можно сказать, типовая. Проектируем специализированное хранилище, ищем в корпоративной среде источники нужной информации, организуем регулярное обновление данных, предоставляем пользователям интерфейс для доступа к данным: непосредственно к хранилищу или к так называемым витринам (*datamart*) и кубам. Наибольшую трудность, кстати, вызывает не сам импорт данных, а поиск в компании людей, которые могут знать, где эти данные взять. Всё-таки 20 тысяч таблиц SAP R3 и примерно столько же в корпоративном хранилище — не шутка, поэтому без хороших проводников раскопки источников информации обернутся поиском иголки в стоге сена.

В случае стационарных рабочих мест этим можно ограничиться. Но специфика ситуации накладывала ограничения на решение.

Предполагалось нетипично большое для аналитического приложения количество пользователей, в потенциале — вся «пехота продавцов» вместе с их непосредственными командирами. Как правило, круг пользователей аналитической БД — десяток специалистов в рабочей группе, а для другой группы организуется новая БД, таким образом распределяется нагрузка, создаваемая тяжёлыми запросами пользователей.

Пользователи должны были иметь возможность работать «в поле», то есть при отсутствии соединения с корпоративной сетью. В связи с их количеством, это требование хоть и усложняло реализацию, накручивая новое звено, но помогало решать вопросы нагрузки децентрализацией обработки.

Как и в любой транснациональной корпорации, необходимо поддерживать многоязычный интерфейс в приложении и непосредственно в данных.

Работа в крупной компании имеет свои особенности. Например, директор информационных систем уровня филиала может и не знать, чем принципиально отличается MS Access от SQL Server. Но показать лицензионную чистоту использования движка Access ему необходимо.

Если с серверной СУБД выбор без особенных затруднений пал на SQL Server, то рабочие места обладали неприятной особенностью: пользователи были лишены прав локального администратора, соответственно, возможности

по установке компонентов системного уровня у них отсутствовали. Встраиваемая в приложение редакция SQL Server Compact 2005 в тот момент не могла быть развёрнута простым копированием. Общего с полноценным SQL Server, даже в его минимальной экспресс-версии, у неё было немного, за исключением названия. По сути, тот же Access, даже местами менее функциональный за счёт отсутствия поддержки временных таблиц, но с проблемами использования вне .NET. На рассмотрение и обход раскиданных граблей в других встраиваемых СУБД времени не было.

С автономным приложением выбор был более очевидным. Стандартной версией .NET на рабочих местах была 1.1, что для 2007 года выглядело устаревшим. Собственно, .NET был нужен для приложения SAP и, в своё время, накатывая на корпоративные компьютеры обновление фреймворка на уровне пакета (*service pack 1*), предприятие столкнулось с проблемами функционирования основного приложения. Поэтому предлагать службе эксплуатации устанавливать .NET 2 было невежливо, мягко говоря. А при анонсированном .NET 3 ещё и неразумно. Так в проекте возникла последняя версия Delphi для Win32, позволяющая строить приложения, развёртываемые простым копированием исполняемых файлов без необходимости установки системных компонентов. Проблемы локализации и автоматического обновления версий уже решались в рамках этой среды и не вызывали вопросов.

Если изобразить получившуюся архитектуру, то получится картинка, представленная на рис. 11.

Построив распределённую архитектуру, мы сталкиваемся с двумя основными проблемами:

- поддержка данных в актуальном состоянии;
- обработка данных относительно слабым, по сравнению с сервером, компьютером пользователя.

Первая проблема была решена односторонней синхронизацией локальных данных с соответствующей их частью в хранилище. Поясню подробнее, каждый пользователь привязан к своей клиентской базе, определяемой географией продаж. Соответственно, торговому агенту с клиентами в Италии нет никакого смысла хранить данные о продажах в Великобритании. Поэтому локальная база данных была намного меньше хранилища и гарантированно не превышала двух гигабайт, которыми ограничен движок Access.

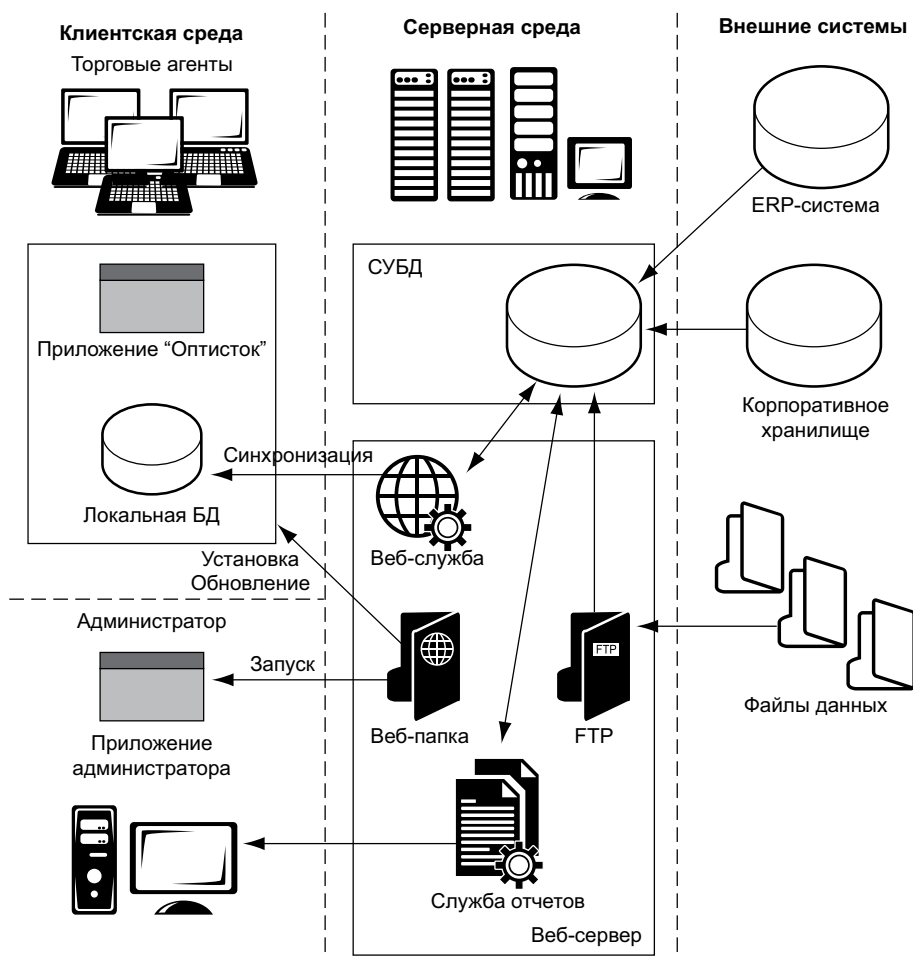


Рис. 11. Архитектура «Оптисток»

Однако передача таких объёмов данных по узким каналам, равно как и вставка в таблицы локального движка миллионов записей, потребовали дополнительной оптимизации. Веб-служба синхронизации была дополнительно нагружена сжатием данных, добавлены функции пакетной передачи, например, по 500 тысяч записей. Вставка в транзакции даже на ADO-технологии позволила довести скорость до 10–20 тысяч записей в секунду, что было достаточным.

Вторая проблема вызывала тревогу у работавших по соседству консультантов, поскольку имевшееся у них в эксплуатации приложение с БД Access размером всего 1 гигабайт проводило за расчётами долгие ночные часы. Но я был априори

уверен, что из этого движка вполне можно выжать необходимую скорость, если не впадать в деструктивный снобизм якобы «ненастоящей СУБД», а подходить к вопросу так же серьёзно, как к обработке на порядок-два больших объёмов на полноценном SQL Server.

Для отладки SQL пришлось в короткий срок соорудить инструмент, подобный Query Analyser, но работавший через ADO с БД Access. В получившем название ADO SQL Tools (MS Access Query Analyzer) и происходила основная разработка расчётов. Время исполнения наиболее тяжёлых пакетов запросов удалось довести до 20–30 секунд на обычном пользовательском ПК или ноутбуке с одноядерным процессором и 1 гигабайтом ОЗУ.

Немало времени было уделено интерфейсу пользователя. Зная, что основная работа по обучению ляжет на его плечи, Поль постарался упростить видимый функционал, внося немало предложений и уточнений в спецификацию. Кроме того, внешний вид приложения, «шкурка»¹, должен был соответствовать корпоративной цветовой гамме, для чего в проект был привлечён профессиональный график. По одежке встречают.

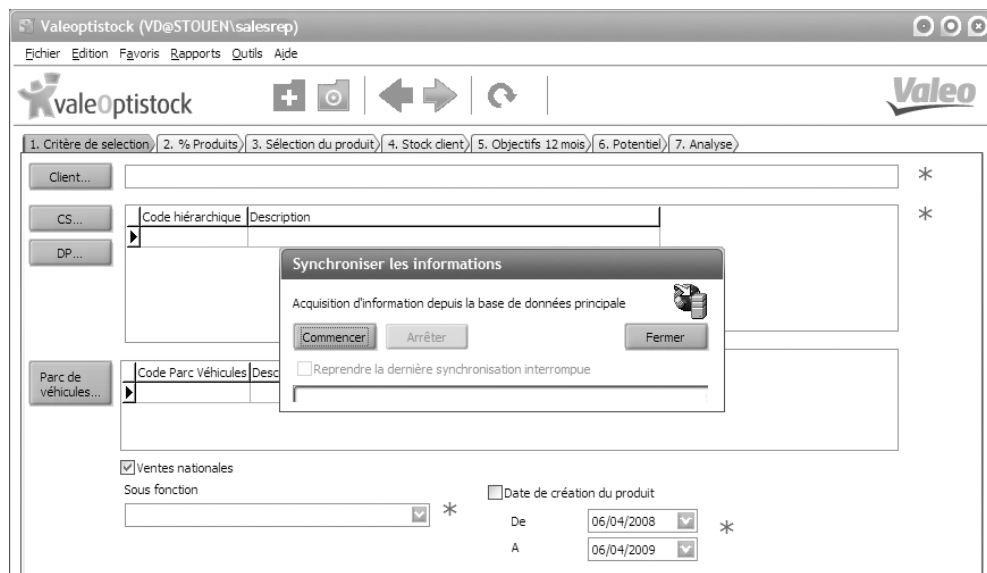


Рис. 12. Внешний вид клиентского приложения «Оптисток»

¹ От англ. skin — изменяемый внешний вид приложения.

Для обеспечения процесса на площадке заказчика пришлось организовывать минимальный набор из трех изолированных сред: разработки, тестирования-приёмки и эксплуатации. С точки зрения 2012 года мощность сервера вызывает улыбку, лишь недавно ему добавили памяти, доведя объём ОЗУ до 4 гигабайт. Но распределённая обработка данных вкупе с использованием множественной обработки на SQL позволила обойтись столь скромными ресурсами.

Добавлю, что корпорация дважды представляла систему на международной автомобильной выставке EquipAuto в 2007 и 2009 годах. Ролик с соответствующим видео и сейчас можно найти на веб-сайте. Неплохой бонус к успешно внедрённому продукту, сделанному при удачной организации процесса и мотивации основных участников.

Архитектура сокрытия проблем

В 2008 году крупный разработчик ERP-системы для розничной торговли оказался в числе наших клиентов. Что же хотела компания с 25-летней историей, двумя сотнями сотрудников и множеством клиентов в Европе и Северной Америке, среди которых такие бренды, как Naf Naf, Vieux Campers, Tally Weijl, Guess, от небольшой программистской фирмы, вчерашнего «стартапа»?

Новая версия системы «по всем правилам архитектуры», подобно каменному цветку, не выходила. Сроки затягивались, приёмки переносились, инвесторы нервничали. Не знаю, кому пришла в голову идея, но фирма решила попробовать в деле так называемую «программную фабрику» (*software factory*). То есть генерировать немалую часть кода и компонентов системы из мета-описаний. Предшествующие внутренние попытки домен-ориентированной разработки на базе .NET, NHibernate и всевозможных лучших практик не приносили результата.

Проведённый Марком, нашим консультантом, уже самый первый аудит во время командировки в Монпелье принёс очень интересную информацию к размышлению. Предыдущая версия была реализована в классической клиент-серверной технологии в среде Microsoft. То есть SQL Server, «толстый клиент» или веб-клиент под ASP.NET с разделяемой на уровне сборок логикой. Все работало прекрасно до некоторого времени. А время, поскольку клиенты крупные, подошло быстро, за несколько лет. База данных разрослась, время отклика как интерактивной работы, так и пакетов увеличилось, производительность

продолжала понемногу деградировать. Например, обработка консолидации результатов торговли за сутки стала превышать эти самые сутки.

Программисты даром времени не теряли и упомянутую консолидацию переписывали три раза. Сначала на SQL, но с курсорами, получив ещё худший результат. Потом попытались избавиться от курсоров, но всё равно не смогли достигнуть требуемой производительности. Потом в команде каким-то образом появились специалисты, сказавшие: «Это не объектный подход!» Они переписали всю обработку на C#, окончательно похоронив производительность модуля.

База данных у клиентов — порядка одного терабайта, с примерно 500 миллионами строк в самой большой таблице, соответствующей истории за несколько лет. В версии-«бабушке» использовались технологии «старой школы» вроде архивации, благодаря чему она как-то справлялась с объёмами, хотя бы и за счёт вывода данных из оперативного контура. Но смена СУБД на более мощную из «большой тройки» и более производительное «железо», видимо, произвела на разработчиков обманчивое впечатление, что всё решится как-нибудь само.

Физическая архитектура хранения, секционирование таблиц, оптимизация индексов и запросов по выборочной статистике — эти вопросы всерьёз не рассматривались, база данных тихо кряхтела одним файлом, правда, на мощном 16-ядерном сервере с 32 гигабайтами памяти и быстрым дисковым массивом.

Разработка между тем шла дальше. Новый толстый клиент работал через слой WCF с сервером приложения. Ведь каждому «специалисту» известно, что если данные в СУБД обрабатываются медленно, то необходимо их оттуда загоня вытаскивать, кэшировать и обрабатывать на сервере приложений. Вдруг, так быстрее получится?

Однажды я наблюдал, правда, уже в другом продукте, как обрабатывали таким образом матрицу прав доступа. Две сотни пользователей, десять миллионов объектов и *voilà* — списки размером порядка 100 миллионов элементов сидят в памяти сервера приложения, где со скрипом тормозов обрабатываются запросами LINQ. Выдавать из СУБД информацию, уже прошедшую необходимые фильтры доступа, почему-то оказалось сложнее.

Вот под это развитие и было решено использовать программную фабрику, чтобы генерировать рутинный код отображения таблиц на объекты, объектов, на их посредников (*proxy*) для доступа через канал WCF и части интерфейса на WPF.

Для опытной разработки нам был предложен относительно новый модуль прогнозирования продаж. В ходе реализации большая часть обработки там все-таки была написана на SQL. Поэтому, несмотря на отсутствие администрирования базы данных и несоответствующую архитектуру хранения, удалось выжать что-то близкое к возможному максимуму. Например, расчёт части прогноза выполнялся несколько секунд. Что оказалось быстрее, чем пропуск результатов через все слои и загрузка отображающих форм WPF, закодированных со всеми «лучшими практиками».

Можно ли, будучи в здравом уме, представить себе, чтобы обработка данных шла быстрее их передачи между слоями системы и отображения? Для этого надо серьёзно постараться в освоении «паттернов», нагромоздить кучу вроде бы правильного, но бессмысленного кода с высокой цикломатической сложностью, глубинами иерархий и связностью классов. Ситуацию не спасала автоматическая генерация кода большей части этого Ада Паттернов по сравнительно простой модели с несколькими десятками сущностей.

Упомянутую проблему консолидации нами было предложено решить с помощью кубов OLAP. Действительно, после такой переработки, процедура стала идти вместо суток несколько часов, причём, что в MOLAP¹, что в ROLAP² вариантах. Что, собственно, означает: при правильной организации физического хранения и грамотном SQL-коде как минимум тех же результатов можно было бы добиться, не выходя за пределы реляционной БД. Например, наше пожелание создать в определённой таблице кластерный индекс не встретило понимания и было потеряно в глубинах архитектурного буйства и организационных процедур.

Дальше наступил ожидаемый поворот. Прежняя консолидация использовалась многими модулями в реляционной форме, и переписывать её на работу с OLAP никто в здравом уме не собирался. Поэтому из кубов OLAP информация перекачивалась обратно в исходную реляционную БД, в таблицы наследуемой структуры. Тем не менее новая, странного вида цепочка процессов РСУБД — OLAP куб — РСУБД всё равно выполнялась быстрее, чем все три варианта консолидации, ранее написанные местными умельцами.

Спустя почти год, мы благополучно закрыли проект и, утерев пот со лба, передали модуль на сопровождение заказчику. К тому времени ситуация до-

¹ От англ. Multidimensional OLAP — многомерная БД во внутреннем оптимизированном формате хранения СУБД.

² От англ. Relational OLAP — использует реляционную СУБД в качестве многомерного хранилища (иерархии таблиц, управляемых мотором OLAP).

шла до попыток внедрения в фирме — продуктовом разработчике — «гибких» экстремальных методик. При наличии штата экспертов предметной области и 25-летнего опыта создания функциональных моделей это означало полный разрыв проектирования с производством.

Через небольшое время фирма была поглощена холдингом Cegid — крупнейшим во Франции поставщиком собственных и приобретённых специализированных отраслевых ERP-решений. Уже имея свои лоскутки по розничной торговле, они просто купили фирму со всеми долгами, потому что сотни крупных клиентов — это не шутка, а серьёзный актив, которому они отныне будут предлагать и свои решения.

Очередной урок, «кейс», экспериментаторам с единственно правильными архитектурами, любителям городить новые слои, чтобы спрятать за ними свою некомпетентность в области СУБД. Не исключаю, конечно, что для некоторых менеджеров, получивших выгоду от поглощения, этот прецедент мог быть и позитивным.

Один из «Технических Дней Microsoft» (TechDays) в 2011 году был целиком посвящён специализации DBA (DataBase Administrator). А выступающий на сцене ведущий эксперт не постеснялся напрямую высказать призыв: «Последние годы я вижу тотальное падение компетенции в области баз данных. DBA, проснитесь!»

Code revision, или Коза кричала

Ревизия программного кода всякий раз напоминает мне эпизод из фильма Г. Данелии «Осенний марафон». Главный герой, преподаватель университета Андрей Бузыкин сидит у своей бывшей сокурсницы Варвары, помогая ей с переводом художественного произведения. Время перевалило за полночь, происходит примерно такой диалог.

— Скажи, Бузыкин, может, я бездарная?

— Не-е-е...

— Но ты же всё повычеркивал!

— Не всё... Но вот это, например, я не мог оставить: «Коза кричала нечеловеческим голосом».

Мой коллега, обладатель диплома историка, переквалифицировавшийся в консультанта по ВІ, как-то посетовал, что он плохой программист. Будучи несколько удивлённым, я успокоил его тем, что в ВІ программирования как такового немного и критичные куски кода всегда могут помочь написать коллеги соответствующей специализации, стоит обратиться к ним по внутренней рассылке. Хуже, когда вполне программистский коллектив умудряется годами работать без системы контроля версий исходников, и тогда в коде половину объёма составляют закомментированные куски многолетней давности. Выбросить их жалко, вдруг пригодятся. Но и контроль версий с архивацией не спасает от цифровой пыли десятилетий. В подобных залежах порой можно обнаружить настоящие образцы софтостроительных антипрактик.

Например, одна ERP-система много лет назад переносилась из файл-серверной архитектуры в среду клиент-серверной СУБД. Вполне ожидаемо в базе данных обнаруживается таблица типа «мегасправочник», хранящая все ссылки вида «ключ-значение». Структура состоит из трех колонок: код справочника, код значения и само значение. В прежней архитектуре ссылочная целостность поддерживалась приложением, теперь же стандартным образом приспособить для этой цели транзакционную СУБД невозможно, потребуется написать достаточно длинный линейный триггер.

Такой универсализм стал причиной использования мегасправочника одновременно для хранения внутренних счётчиков нумерации записей: текущая величина хранилась в строковом поле колонки «Значение» в формате «префикс; текущий номер». Приложение считывает текущее значение счётчика, анализирует строку, выделяя префикс и величину, переводит величину из строки в целое, увеличивает его на 1, формирует новое значение строки и снова записывает всё это обратно в базу данных.

Кроме перечисленных манипуляций со строкой, вначале делается попытка заблокировать запись через соответствующую опцию SQL-запроса. Мысль правильная, но, к сожалению, блокировка делается вне контекста транзакции, то есть снимается сразу после окончания выполнения запроса. На вопрос: «У вас конфликтов нарушения первичного ключа не было?» был дан самый оригинальный ответ за всю мою практику: «Они нам мешали делать каскадные обновления в связанных таблицах, и мы их удалили, оставив просто индексы».

В другом случае на форме Delphi-приложения имелась группа из двух опций (радиокнопок) для взаимоисключающего выбора. Кнопки были подписаны как «Объём ограничен» и «Объём неограничен». Вроде бы ничего особенного. Но открываем форму и обнаруживаем, что кнопка с надписью «Объём **ограничен**»

поименована программистом как «КнопкаОбНеограничен». И, разумеется, наоборот. Ошибся человек, бывает...

К счастью, в коде формы есть только одно место, где значения кнопок используются. Видимо, во избежание путаницы процедура оформлена следующим образом:

```
var ОбъемТакиОграничен: boolean;  
...  
if КнопкаОбНеограничен.Выбрана then  
  ОбъемТакиОграничен := true  
else  
  ОбъемТакиОграничен := false;  
...  
ВызовКакойТоФункции(ОбъемТакиОграничен);
```

Дальше ревизия коснулась SQL-кода. Программист пытался выбрать следующий элемент списка, обрабатывая только первую запись из пришедшего по запросу набора. При этом сортировку он делал совсем по другой колонке, нежели порядковый номер в списке. В итоге выбиралось что угодно, но не следующий элемент.

Не буду утомлять вас другими примерами, надеюсь, вы просто поверите в их многочисленность и оригинальность. Мне хотелось лишь донести простую мысль, что ревизия кода, несомненно, весьма полезная процедура, но как минимум при двух условиях:

- эта процедура регулярная и запускается с момента написания самых первых тысяч строк;
- процедуру проводят специалисты, имеющие представление о системе в целом. Потому что отловить бесполезную цепочку условных переходов может и компилятор, а вот как отсутствие контекста транзакции в обработке повлияет на результат, определит только опытный программист.

Дж. Фокс [2] выводит из своего опыта проектной работы в IBM важную мысль, что большой ошибкой является привлечение к процессу внутреннего тестирования и обеспечения качества посредственных программистов. По его мнению, компетентность специалиста в этом процессе должна быть не ниже архитектора соответствующей подсистемы. Действительно, ведь оба работают примерно на одном уровне, просто один занят анализом, а другой — синтезом.

Качество кода во многом зависит от степени повторного использования, поэтому приведу простой и доступный способ проверки того, не занимается ли ваша команда программистов копированием готовых кусков вместо их факторизации. Для этого регулярно делайте сжатый архив исходников, например zip с обычным коэффициентом компрессии, и оценивайте динамику роста его размера относительно количества строк. Если размер архива растёт медленнее, чем количество строк, это означает рост размера кода за счёт его копирования.

Наживулька или гибкость?

Приходишь в отечественную компанию,
смотришь, как у нее устроено IT, и видишь,
что люди просто упали с дуба.

М. Донской, из интервью

Не все гигагерцы и гигабайты расходуются впустую. Кризис в софтостроении, о котором говорят уже более 30 лет, продолжается. В ответ на усложняющиеся требования к программным системам и неадекватные им методологии (технологии), особенно в части моделирования и проектирования, индустрия выставила свое решение. Оно состоит в достижении максимальной гибкости средств программирования и минимизации ошибок кодирования. Проще говоря, если мы не можем или не успеваем (что в итоге приводит к одному и тому же результату) достаточно хорошо спроектировать систему, значит, надо дать возможность быстро и с минимальными затратами её изменять на этапе кодирования. Но принцип для заказчика остался прежним: «Быстро, качественно, дешево — выбери два критерия из трёх».

Поднимать тему так называемой гибкой (*agile*) разработки программ достаточно рискованно. С одной стороны, вокруг сюжета много шума, эмоций и мало объективной фактической информации. С другой — большинство встречавшихся мне собеседников опирались больше на веру, чем на рациональные аргументы, что делало дискуссию бессмысленной. Но мы тем не менее попробуем.

Причина возникновения экстремальных методик, сосредоточенных на фазе кодирования, не случайна. Американский специалист по методологиям

софтостроения Кэпер Джонс в своей книге[23] приводит весьма удручающие статистические данные, например:

- среди проектов с объёмом кода от 1 до 10 миллиона строк только 13 % завершаются в срок, а около 60 % свёртываются без результата;
- в проектах от 100 тысяч до 1 миллиона строк эти показатели выглядят лучше (примерно 25 % и 45 %), но признать их удовлетворительными никак нельзя;
- в проектах примерно от 100 тысяч строк на кодирование уходит около 20 % всего времени, и эта доля снижается с ростом сложности, тогда как обнаружение и исправление ошибок требует от 35 % времени с тенденцией к увеличению.

В любом софтостроительном процессе, будь то заказной проект или продукт для рынка, всегда можно выделить 4 основные стадии:

- анализ, чтобы понять «что делать»;
- проектирование, чтобы определить и запланировать «как делать»;
- разработка, чтобы собственно сделать;
- стабилизация, чтобы зафиксировать результат предыдущих этапов.

Если стадии, органично совпадающие с концептуальным, логическим и физическим дизайном системы, расположить иерархически без обратных связей, то получим классическую схему «водопад», исторически считающуюся первой методологией в софтостроении.

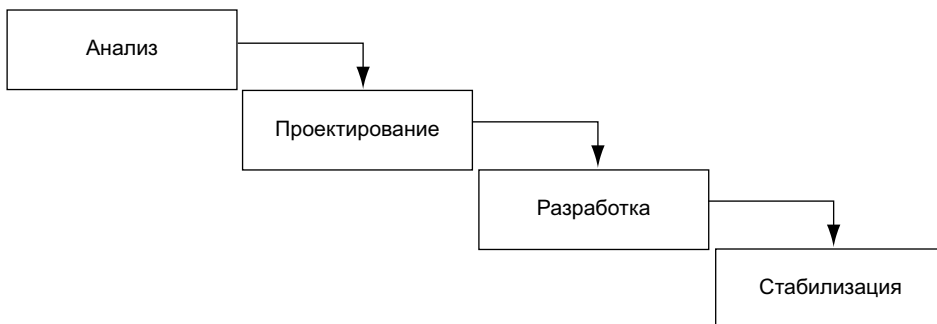


Рис. 13. Методология «водопад» в идеальном случае

«Водопад» также соответствует разработке «сверху-вниз» в структурном программировании: выделяем самые общие функции системы, проводим их декомпозицию на подфункции, а те, в свою очередь, на под-подфункции и так далее, пока не упрёмся в элементарные операции.

Что же не так в схеме?

С увеличением сложности реализуемой системы анализ и следующее за ним проектирование начинают занимать всё больше времени. Постепенно обнаруживаются новые детали и подробности, изменяются требования к системе, возникают новые сопутствующие задачи, расширяющие периметр. Приходится, не отдавая проектную документацию в разработку, возвращаться к анализу. Возникает риск заикливания процесса без конечного выхода какого-либо программного обеспечения вообще.

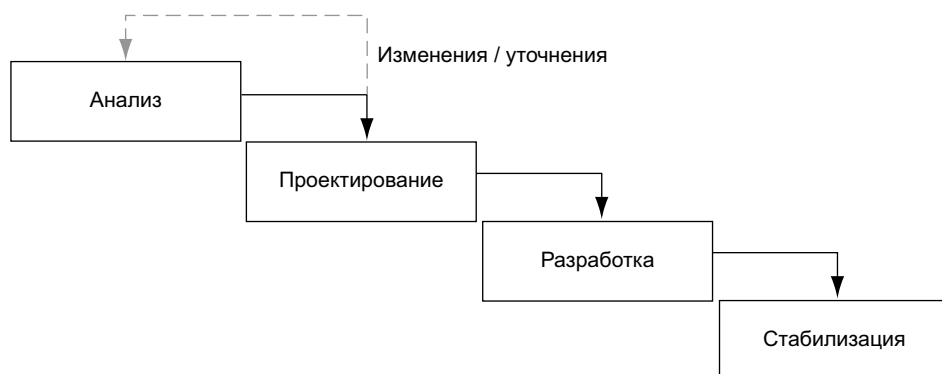


Рис. 14. На практике «водопад» заикливается с увеличением сложности проекта

Из сказанного вовсе не следует, что методология плоха. Просто она имеет свои границы применения, широта которых напрямую зависит не только от опыта аналитиков и проектировщиков, но и от новизны моделируемой предметной области. В достаточно консервативных банковских или промышленных приложениях «водопад» может подойти и для комплексной системы. Если же, например, возникает принципиально новый рынок, то найти опытных подрядчиков проблематично, а бизнес-направление заказчика находится в постоянной реорганизации, поэтому стадия анализа, занимающая больше 3–4 недель, рискует быть малопродуктивной. И не только стадия анализа, но и весь проект в целом. Тут впору подумать над временной автоматизацией в рамках офисного пакета и скриптовых сред вместо комплексного решения.

Очень важно отделить редкую ситуацию **«бизнес меняется еженедельно»** от гораздо более распространённой **«представления команды разработчиков о бизнесе меняются еженедельно»**. Если вам говорят о якобы часто изменяющихся требованиях, всегда уточняйте, о чём, собственно, идёт речь.

Но если проблема заикливания на требованиях может быть успешно решена выбором подрядчиков, уже имевших опыт в построении систем данного типа, то другая, гораздо более значимая проблема несовпадения взглядов заказчика и подрядчика на казалось бы одни и те же вещи стабильно проявляется с ростом проекта. Даже если принять во внимание, что только 20 % требований специфичны для заказчика, тогда как 80 % исходят непосредственно от предметной области инвариантно среде и контексту, то эти 20 % способны угробить весь проект.

Для снижения рисков такого рода в конце 1980-х годов была предложена спиральная модель софтверного строительства.

Необходимо отличать **спиральную** модель от **итеративной**. Спиральная модель сходится в точку «система готова», итеративная модель в общем случае не сходится, но обеспечивает реализацию всё новых и новых требований.

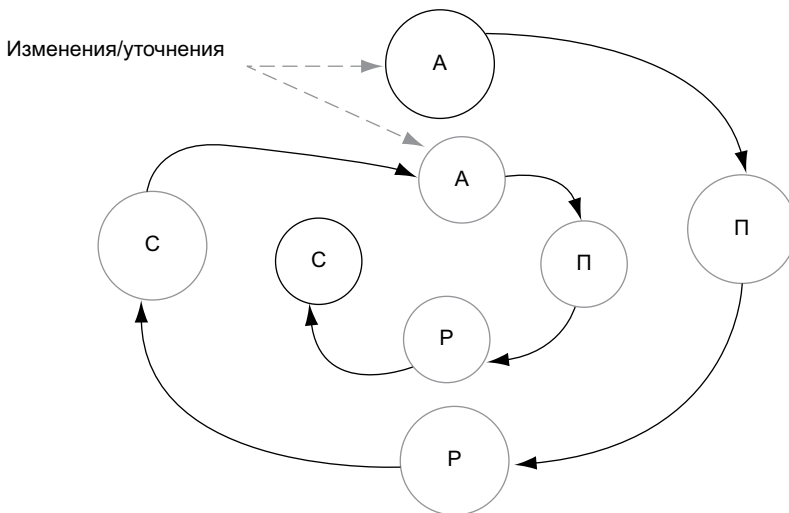


Рис. 15. Спиральная модель с двумя витками

Ключевой особенностью спиральной технологии является *прототипирование*. В конце каждого витка после этапа стабилизации заказчик получает в своё распоряжение ограниченно работающий прототип целой системы, а не отдельных функций. Основная цель прототипа состоит в максимально возможном сближении взглядов заказчика и подрядчика на систему в целом и выявлении противоречивых требований.

Спиральная модель не навязывает присутствие всех стадий на каждом витке. Вполне может статься, что первый же прототип будет удачным, а функциональная и техническая архитектуры соответствуют требованиям. Тогда финальные витки будут фактически состоять только из разработки и стабилизации.

Слабым звеном в спиральной методологии является определение длительности очередного витка, его стадий и соответствующее выработанному плану управление ресурсами. Пока анализ не выдал концептуальные модели, проектировщики и ведущие программисты ограничены техническими требованиями, тогда как рядовые программисты просто ожидают спецификации. Но неудачно сократив фазу анализа или проектирования на первом витке, тем самым можно увеличить их общее количество, рискуя выйти за рамки первоначальной оценки сроков и бюджета.

Тотальный анализ и проектирование вырождают спираль в водопад, тогда как формальный подход, «для галочки», выдающий бесполезные для программистов спецификации, превращает спираль в бесконечный цикл, прерываемый управленческим решением. Поиск компромисса в такой ситуации — трудная задача многокритериального выбора, в большой степени зависящая от опыта и здравого смысла руководителей рабочих групп и проекта в целом.

Таким образом, из научной плоскости мы переходим в область экспертных оценок. Поэтому слово «методология» справедливо вызывает у многих негативное отношение. Правильнее было бы говорить о технологии ведения софстроительных проектов, но исторически появившийся термин так просто из обращения не выкинуть. Да и надо ли?

Создатели итерационных методологий, также называющихся гибкими (*agile*) или экстремальными, выкинули не термин, а фазы анализа и проектирования.

Ключевой особенностью гибкой методики является наличие мифологического титана — владельца продукта (*product owner*), который лучше всех знает, что должно получиться в итоге. На самом деле это просто иная формулировка старого правила «кто платит, тот и заказывает музыку». Именно владелец, за рамками собственно гибкого процесса, гением своего разума проводит анализ и функциональное проектирование, подавая команде на вход уже готовые пачки

требований. Размер пачки должен укладываться в интеллектуальные и технологические возможности разработчиков, которым предстоит осуществить её реализацию за одну итерацию.

В итоге мы получаем знакомую софстроительную схему «снизу-вверх», появившуюся на свет гораздо раньше «водопада», с его упорядочивающей моделью «сверху-вниз». То есть мы не знаем точно, что хотим получить в целом, но знаем отдельные функции, реализовав которые, мы, возможно, придём к решению.

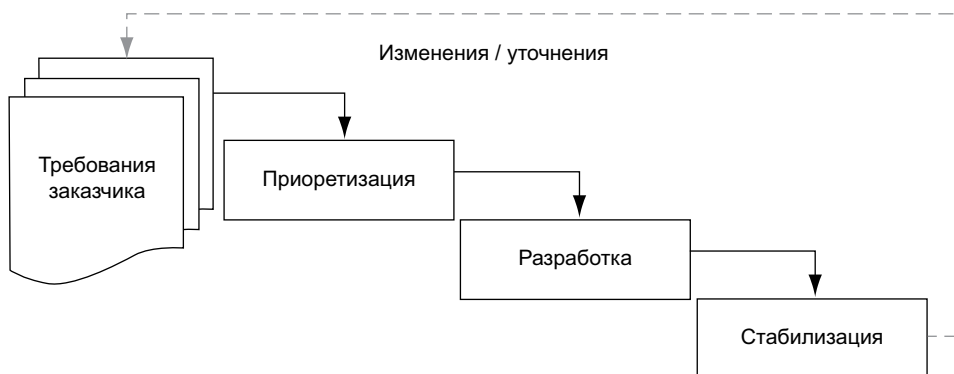


Рис. 16. «Гибкая» методология соответствует классической схеме «снизу-вверх»

С точки зрения проектирования такой подход даже хуже обсуждавшихся прецедентов в UML. Прецеденты использования, «кейсы», хоть как-то формализуются, накапливаются и обобщаются на стадии, формально отсутствующей в гибкой разработке. Поэтому новое требование-прецедент, поступившее на вход, «обобщается» с прежними уже на стадии разработки, приводя к необходимости серьёзной реструктуризации (рефакторинга) кода программ и подпорок из тестов.

Не буду заниматься критикой, её можно без труда найти даже в энциклопедических статьях. Поговорим лучше о некоторых позитивных моментах, которые вносит гибкая методология.

Для программистов позитив, к сожалению, изрядно разбавлен издержками производственного процесса. Методология, по сути, направлена на увеличение времени работы с клавиатурой и не располагает к размышлениям. Пиши код! Поэтому для стимуляции персонала процесс окружен религиозной атрибутикой, манипуляциями, иносказаниями и метафорами. С другой стороны, требования к уровню программиста ограничиваются знанием конкретных технологий ко-

дирования, стандартных фреймворков, «умением разбираться в чужом коде» и «умением работать в команде», уже упоминавшимся в словаре для начинающего соискателя. Способность решать олимпиадные задачки здесь от вас не требуется. Скорее, наоборот, будет помехой.

Позитив для заказчика в том, что, осознавая свою несхожесть с мифологическим титаном мысли, он может достаточно быстро увидеть сформулированные требования и сценарии в реализации, отлитыми, разумеется, не в бетоне, а в гипсе, и на практике понять их противоречивость и неполноту. После чего он может переформулировать существующие и добавлять новые требования с учётом уже набитых шишек. Тем не менее с ростом сложности системы возрастает и риск увеличения стоимости внесения изменений. И если проект выходит за рамки бюджета, то «козлом отпущения» становится именно владелец продукта.

Другой позитив для компании-заказчика состоит в непосредственной близости выполняемой подрядчиком работы. Зачастую «гибкие» команды работают на площадке компании и доступны в любой рабочий момент. Если заинтересованному лицу не хватает информации, он может просто подойти и посмотреть на месте, поговорить с исполнителем и тем самым восстановить расстроившееся было душевное равновесие.

Ничего не напоминает? Всего 15–20 лет назад эти же подрядчики сидели на тех же площадках, но назывались «отделами АСУ» и входили в штат фирмы. Да, теперь сторонние программисты обходятся вдвое-втрое дороже прежних сотрудников, зато в любой момент можно устроить имитацию конкурсного отбора исполнителя, а при необходимости быстро свернуть непрофильную деятельность, не испытывая законодательных и профсоюзных затруднений с сокращением персонала.

Позитив для подрядчика состоит в том, что «гибкая» разработка позволяет вовлечь в проект как можно больше разработчиков с менее высокими требованиями к квалификации. Это позволяет содержать больше сотрудников в штате, включая оффшорные команды. Будучи поставленным перед выбором между небольшой программистской фирмой с квалифицированным персоналом и софтверхаузом-«тысячником», крупный заказчик в общем случае склонится ко второму варианту.

Как вы заметили, я начал с «водопада» и завершил «гибкой разработкой». Хотя адепты обоих подходов испытывают друг к другу сложные чувства, не бросилось ли вам в глаза их разительное сходство? Обе методологии закливаются с увеличением сложности проекта. Только «водопад» замыкается на тотальном анализе и проектировании, а гибкая методика «уходит в себя»

на разработке и стабилизации, что, в общем, даже не скрывается, а рисуется на слайдах презентаций. Из этого следует вывод, что успешно выполненный в «водопадной» схеме проект может быть также в большинстве случаев выполнен в «гибкой» разработке. И наоборот, если не касаться вопроса людских ресурсов. Дело в масштабе.

Труднее формально определить пороговую сложность системы, за которой начинаются проблемы. Я обозначил бы её как систему, которую даже один опытный аналитик способен охватить формальными моделями за относительно короткий срок, исчисляемый несколькими неделями. Например, заказной, то есть нетиражный, пакет для расчёта зарплаты, подсистема формирования и массовой рассылки счетов клиентам или система складского учёта в компаниях среднего и крупного масштаба.

Как происходит заикливание даже в простых случаях? Программисты классифицируют коров и столы по признаку наличия четырёх ножек, после чего всю энергию тратят на то, чтобы написать интерсепторы, аспекты, применяют мощные инструменты рефакторинга кода для того, чтобы *ad hoc*¹ разрешить некоторые возникающие противоречия в созданной модели.

Конечно, это просто шутка, в которой немало правды. В реальности же наиболее распространённым явлением становится частичное дублирование структур и функциональности отдельных подсистем, реализуемых разными командами и владельцами. Поскольку общего взгляда на систему нет.

Борьба с заикливанием с обеих сторон нередко принимает причудливые формы.

Совсем недавно мне выдалось консультировать по сугубо техническим вопросам одну скрам²-команду. Проект был заказан крупным автопроизводителем и касался разработки бортовой системы мониторинга и управления периферийным оборудованием. Функциональные спецификации составили вместе толстую пачку листов формата А4, напечатанных с двух сторон, думаю, в общей сложности не менее 2 тысяч страниц. Понятно, что никто из программистов в здравом уме не стал читать документацию целиком, а взяли несколько функций, под которые и начали строить реализацию. После четырёх месяцев работы выяснилось, что архитектура эволюционным путём не выстраивается, хотя заказчик регулярно видел разные красивые экраны с заглушками и симу-

¹ От лат. *ad hoc* — «по месту», решение конкретной проблемы, не предназначенное для какого-либо обобщения.

² От англ. *scrum* — одна из активно продвигаемых методик гибкого софтверного строительства.

ляцией приходящих от устройств прерываний. Разумеется, и я не стал читать все эти тысячи страниц, ограничившись весьма интересным документом, содержащим иерархию функций, то есть фактически глоссарий функциональной декомпозиции системы. Из документа следовало, что архитектура, состоящая из набора служб, доступных на общей шине (здравствуй, CORBA), охватывала несколько верхних уровней иерархии. Однако такая перестановка означала переделку большей части системы, тогда как регламент не разрешал увеличить время очередной итерации до минимально необходимых 2–3 месяцев, а бюджет и ресурсы не позволяли начать параллельную стройку. В итоге команда осталась в прежней архитектуре, осознавая на собственной шкуре, что затраты на добавление новых функций растут.

Совсем свежий пример: настоящее время, крупная корпорация — строится внутренняя система управления предприятием. Официально написаны 3,5 тысячи страниц функциональных спецификаций, полтора десятка программистов в том же «скраме» уже приступили к реализации отдельных частей. Через год-полтора будет ясно, получилось ли что-нибудь в итоге.

Эти два примера вполне соответствуют тенденциям взаимного перекладывания ответственности на сложных проектах. Заказчик осознаёт, что реализовать спецификации своими силами невозможно, прежде всего потому, что при таком объёме они тем не менее неполные и неизбежно содержат противоречия. Подрядчику же в принципе наплевать на спецификации, он будет крутить итерации, честно реализуя заявленный функционал и отрабатывая бюджет. Получился коровник на подпорках с покосившимися заборами и дырявой крышей вместо современного агрокомплекса? Извините, всё по спецификации, каждые две-три недели вы видели расцвеченные фотографии разных участков возводимого сооружения.

Синтез «водопада» сложной системы, итоги проектирования которого подаются на вход «гибкой» производственной машины кодирования и стабилизации — что может быть бессмысленнее и беспощаднее?

Кроме частных примеров относительно крупных заказов, современная тенденция — огромное число некритичных проектов, программ и систем-пристроек к основной КИС. Масштаб проектов небольшой (сотни тысяч строк кода), заказчик точно не знает, что хочет получить в итоге, а подрядчик не имеет опыта в данной предметной области, если вообще имеет хоть в какой-то, и поэтому не может ему объяснить, что кактусы за полярным кругом не растут. Для такой ситуации привлечение команд с имеющимся требуемым опытом квалифицированными специалистами и технологиями предметно-ориентированных языков или

разработки по моделям маловероятна, поэтому пусть уж лучше итеративная методология «наживульки», чем никакая, как оно зачастую бывало в эпоху штатных отделов АСУ.

Тесты и практика продуктового софтостроения

Привожу комментарий Максима Крамаренко, руководителя компании Trackstudio, выпускающей одноимённый продукт для управления задачами в софтостроении.

У нас тотальные модульные тесты, что называется, «не пошли». Сложилось впечатление, что их хорошо использовать для продуктов, которые реализуют какой-то стандарт или спецификацию (СУБД, веб-сервер), но для тиражируемого веб-приложения это смертельно. Причины:

1. При изменении спецификаций затраты времени на приведение в актуальное состояния тестов могут быть куда больше, чем на собственно код. Скажем, если пользователи захотят поменять синтаксис SQL в СУБД и писать SEARCH вместо SELECT, то это одно изменение продукта в одном месте приведёт к переписыванию почти всех тестов. Если для СУБД такие пожелания пользователей — редкость, то для менее стандартных программ — обычное дело.
2. Сбои, которые могут выловить тесты (повторяемый сбой в модуле, ранее уже исправлявшийся), — довольно редкое дело. Гораздо чаще сбои возникают в интеграции разных технологий, которые сложно автоматически протестировать. Например, при работе под таким-то браузером при таких-то настройках вот этот JavaScript работает неправильно.
3. Наличие модульных тестов сильно затрудняет масштабный рефакторинг. Если у нашего приложения есть какой-то внешний API, то все, что ниже, мы можем менять быстро и без особых проблем. Но если для этого низкоуровневого кода есть тесты, то их придётся основательно переделывать. (Прим. автора: при этом функциональные тесты, работающие с API, переделывать не требуется.)
4. Если не напрягаться с выпуском раз в 2 недели, то возможность быстро что-то протестировать не так уж и важна. Мы себя совер-

шенно нормально чувствуем с испытанием бета-версии в течение 2–3 месяцев, зачем чаще?

5. Одни из наших конкурентов широко используют agile-методы и TDD¹, но что-то оно им не очень помогает писать безошибочный код. Мы сравнивали количество найденных проблем в течение месяца-двух после *major release*, у нас показатели лучше в разы, если не на порядок. Частый выпуск версий просто не позволяет им довести код до ума и провоцирует исправление старых и серьёзных проблем методом написания «залепени». (Прим. автора: исправление ошибок, добавляющее новые проблемы.)
6. Я совсем не уверен, что пользователи хотят получать новую версию раз в 2 недели. TrackStudio 3.5 вышла примерно через полгода, после TrackStudio 3.2, и мы получили много негативных откликов, что такие частые релизы заставляют их обновлять локальную документацию и задерживают их собственные процессы на несколько месяцев. Многие пользователи совершенно спокойно живут без обновлений несколько лет. Если они купили продукт — значит, он делает то, что они хотят. Если чего-то не делает — они все равно уже нашли *workaround* (обходное решение), им всё равно.
7. Для корпоративного софта пользователи хотят длительного периода поддержки, пара лет, минимум. Если мы выпускаем продукт раз в 2 недели и находится серьёзная ошибка в версии годичной давности, то нам её нужно будет исправить примерно в 40 ветках. Конечно, править 40 веток кода никто не будет, пользователям сообщат, что ошибка будет исправлена в следующей версии, многие пользователи перейти на неё не смогут (см. п.6) к большой радости конкурентов с предложениями типа *competitive upgrade offer*.

Максим затронул важную тему модульного тестирования, являющуюся краеугольным камнем всех гибких методик: если изначально неизвестно, что выстроится в итоге, дом или коровник, то подпорки у его стен должны быть в любом случае.

¹ От англ. *Test Driven Development* — разработка тестирующего кода одновременно или ранее кода тестируемого.

Излишне религиозная атмосфера превратила вполне здравую и работающую с 1970-х годов технологию модульных тестов в настоящий карго-культ. Подобно жителям островов Меланезии, старательно строящих соломенные аэропланы для привлечения сбрасывающих груз транспортных самолётов, адепты 100 % покрытия кода модульными тестами считают, что это обеспечит успех проекту.

Разработка модульных тестов — это тоже разработка. Для 100 % покрытия потребуется примерно столько же времени, сколько и на основную работу. А может, и больше, смотря как подойти к делу. По моим наблюдениям, соотношение объёмов рабочего и тестирующего кода примерно 1 к 2.

В отличие от тестов функциональных, завязанных на интерфейсы подсистем, модульные тесты требуют переработки одновременно с рефакторингом рабочего кода. Это увеличивает время на внесение изменений и ограничивает их масштаб, приучая разработчика **минимизировать реструктуризацию**, заменяя её надстройкой, быстро трансформирующей архитектуру в Ад Паттернов.

Модульные тесты тоже бывают сложными, а значит, с высокой вероятностью могут содержать ошибки. Тогда возникает дилемма: оставить всё как есть или перейти к мета-тестированию, то есть создавать тест для теста.

Наконец, модульный тест — это самый нижний уровень проверок. Прохождение серии модульных тестов вовсе не гарантирует, что пройдут функциональные тесты.

Практические выводы. Соизмеряйте затраты на создание и поддержку автоматизированных модульных тестов с бюджетом проекта и располагаемым временем. Тем более плохо фанатично навязывать разработку «от тестов», умалчивая о названных особенностях и не учитывая другие возможности. Например, код, генерируемый по моделям, вообще не требует модульных тестов.

Говорящие изменения в MSF и выключатель

Мне с давних пор импонирует подход, описанный в MSF¹. Прежде всего масштабируемостью, относительной логичностью, разумными ограничениями на совмещение ролей в проекте. Но целью написания этих строк вовсе не является

¹ Microsoft Solution Framework — методология софтостроения, опирающаяся на практики Microsoft.

продвижение технологий известной всем корпорации, думаю, и без меня для этих целей найдутся хорошо оплачиваемые консультанты.

Интерес представляет следующий факт: в своей текущей редакции MSF 4.0 была разделена на два направления: MSF for Agile Software Development и MSF for CMMI Process Improvement.

CMMI, или Capability Maturity Model Integration, — модель зрелости процессов организации вообще и софтостроения в частности. Ключевое слово здесь именно «зрелость», описываемая в CMMI несколькими уровнями организации: от хаоса до системы качества.

Соответственно, одно из нынешних направлений MSF предназначено для достижения зрелости процессов софтостроения или дальнейшего улучшения процессов уже более-менее зрелых организаций. А второе... для гибкой разработки.

В завершение темы было бы непростительно не вспомнить о производственной системе Toyota, которую почему-то считают основой того же скрама. Ключевой особенностью системы в Тойоте является принцип «дзидока» (*jidoka*), означающий самостоятельность людей в управлении автоматизированной производственной линией. Если рабочий видит нарушение качества продукции или хода процесса, он имеет право, повернув соответствующий рубильник, остановить всю линию до установления причин дефектов и их устранения.

Внедрение таких методик вполне возможно и вне рамок японского общества. New United Motor Manufacturing Inc (NUMMI) — знаменитое совместное предприятие Toyota и General Motors, ещё в 1970-х годах вошло в «кейсы» бизнес-школ как пример повышения эффективности и качества через смену культуры работы.

Качество программного продукта — многозначное и сложное понятие. Производственная культура — ещё более сложное. В одном можно быть уверенным: ни о какой культуре софтостроения не может идти и речи, если любой программист из коллектива не способен остановить бессмысленный циклический процесс для выяснения, какого же рожна по историям заказчика потребовалось обобщать четырёхногих коров и обеденные столы.

Приключения с TFS

Как вы заметили, я стараюсь перемежать темы, несколько напрягающие мыслительные органы, с занимательными зарисовками из жизни. Продолжим традицию историей о совместимости компонентов сложной тиражируемой системы.

К концу одной недели установка системы под названием MS Team Foundation Server 2008 на виртуальном сервере с Windows 2008 и SQL Server 2005 ещё не завершилась, но пользоваться репозиторием уже было можно.

Все началось в предшествующую пятницу. В процессе было задействовано несколько человек и от клиента, и от нашей команды, включая меня по части установки компонентов СУБД. На неделю практически парализована всякая деятельность, ещё не возобновившаяся в полной мере, до сих пор работает только базовый функционал.

Заклинания, «гугление», помощь консультантов по TFS и даже переписка с Microsoft применялись многократно. Моё осторожное предложение «установить за пару часов SVN для небольшой бригады», высказанное перед началом этой «операции «BI», не встретило понимания. Тогда как использовать систему предполагалось практически только для работы с исходниками, то есть функционал управления задачами не востребован.

Заказчик, в лице своих администраторов, кстати позитивно отнёсшийся к установке SVN — крупная корпорация национального уровня со своей сложившейся внутренней инфраструктурой и несколькими уровнями бюрократии в эксплуатационном секторе, что означает: все решения принимаются медленно и проходят длительные согласования. Но команда привыкла работать в TFS.

Второе отступление по поводу требуемых компонентов. Собственно, TFS 2008, как выяснилось, требует для себя:

- SQL Server 2005 или выше;
- Reporting Services 2005 или выше;
- Analysis Services 2005 или выше;
- Sharepoint Services не ниже 3 SP1;
- Internet Information Server 6 или выше;
- .NET 3.5 SP1;
- ещё что-то по мелочи вроде конкретных хотфиксов.

По мере изучения сего списка я постепенно впадал в прострацию, сравнивая с ничего не требующим пакетом SVN.

Версия MS SQL Server 2005, поскольку 2008 клиентом не эксплуатируется и даже ещё не куплена. Гетерогенная сеть хоть и интегрируется с Active Directory,

но не до конца: интегрированная безопасность (*integrated security*) для клиентов не работает, так как они входят в сеть под профилем NetWare. Ну, и ладно бы с этим, ведь создать регистрации на небольшую группу в TFS несложно, как и в SVN. Плохо другое, доступ с компьютера либо в Интернет с аутентификацией через прокси, либо в локальную сеть к серверам. То есть работать в режиме переключения с терминала на Интернет в браузере с обычного рабочего места нельзя, эта опция доступна только с компьютеров администраторов в отделе поддержки. В итоге все манипуляции проходят именно там в паре с сотрудниками компании.

Изначально под TFS был выделен виртуальный сервер с Win 2008, но базы данных необходимо располагать на другом сервере в обслуживаемом поддержкой кластере.

Предлагаю использовать виртуальный сервер с Win 2003, чтобы избежать возможных проблем с 2005-ми версиями программ. Снова не встречаю понимания, так как скопировать образ — лениво, да и вообще надо переходить на новые технологии.

Через полтора дня ко мне обращаются за консультацией по этому поводу.

Устанавливаем службу отчётов (*Reporting Service*). У 2005-й версии Reporting Services на 2008 сервере есть проблемы с Internet Information Server 7. Дополнительно устанавливаем совместимость с IIS 6. Базу данных на удалённом сервере программа установки создаёт. Танцуем с бубном вокруг создания Identity, но кроме как в Default Application Pool сервис не ставится. Танцы через несколько часов прекращаем, остаётся в этом пуле.

Убеждаюсь что установщик TFS не находит службу аналитики (*Analysis Service*), иду читать документацию. Ребята безуспешно продолжают искать в гугле «код для прохода на следующий уровень игры». Вскоре вычитываю, что Analysis Services должны находиться на том же сервере, что и базы данных.

Служба там не установлена, так как сервер в кластере был выделен под базы данных. Ставить дополнительно Analysis Services на эксплуатационный кластер никто в здравом уме не хочет. Администратор пишет письмо Microsoft. Их поддержка изящно уходит от прямого ответа, говоря, что Analysis Services в кластере да ещё и рядом с эксплуатационной СУБД — это не вполне хорошо. Но в то же время говорит, что теоретически можно поставить и на разные серверы.

Только в официальной документации об этом не написано, нужен консультант, желательно от самого Microsoft. Вызов консультанта с предварительным утверждением бюджета — это ещё неделя простоя. Поэтому вариант отвергается.

Принято решение ставить все на один виртуальный сервер. Администраторы кисло морщат лбы, потому что кластер уже настроен под регулярные процедуры поддержки баз данных, а так они получают ещё один сервер в обслуживание, хоть и виртуальный. Но нехотя соглашаются.

Переустанавливаем на него: SQL Server, Reporting Service, Analysis Services. СУБД и все компоненты настроены и работают. Ухожу заниматься собственно делами проекта.

Вскоре у коллег начинаются новые проблемы: не ставится TFS с неким сообщением об ошибке, на которое в гугле находится всего 2 записи, и кода прохождения этого уровня занимательной игры в них нет.

Начинается переписка с экспертами по TFS уже нашей родной конторы. Находим, что вручную и строго предварительно нужно было ставить Sharepoint не ниже 3.0 Service Pack 1. Потому что под Windows 2008 Server установщик TFS этого не делает. Выясняется, что в Windows 2008 Server стандартной версии при попытке добавить роль AppServer искомые службы Sharepoint в списке отсутствуют. Надо её скачивать и устанавливать. Что и было в итоге сделано.

Чуть позже обнаружилось, что при инсталляции служб Sharepoint был выбран полный режим, что неправильно, потому что при этом установщик ставит собственный экземпляр SQL Server Express, игнорируя уже имеющийся на сервере.

Сносим службы Sharepoint, но оказывается, что созданный экземпляр SQL Server инсталлятор Sharepoint не удаляет. Ну что же, удаляем сервис руками, чистим каталоги и реестр.

Дальше работа возобновляется без моего участия, и к вечеру четверга приходит радостная весть: TFS встал, можно попытаться к нему присоединиться.

У всех пользователей похожие имена типа `tfs_userNN`. На моей локальной виртуальной машине есть проблема: не могу добавить файлы в репозиторий. Права у всех одинаковые. Сообщение об ошибке, связанное с «*cloaked folder*», также в гугле встречается редко. Ищем корень проблем в виртуальной машине (VirtualBox вместо VPC 2007), потом в виртуальном диске — проекции (*subst*) директория. Подозрения не подтверждаются. Проходит полдня, создаётся новый пользователь с аналогичными правами. Заработало, но причины так и остались неясны.

В этот момент администраторы прописывают имя нашего сервера в корпоративной DNS¹, теперь можно отказаться от использования прямого адреса.

¹ От англ. Domain Name System — система доменных имён, в сети TCP/IP позволяет обращаться к серверам по имени.

Первое же действие выдаёт окошко регистрации, которая проходит нормально, но все последующие операции дают ошибку 401 not authorized.

На удивлённый вопрос «Что случилось?» от администраторов приходит «письмо счастья» с прикреплённым файлом конфигурации для *proxy*, который надо поместить в корневой каталог локального (!) Internet Information Server, вот такая неочевидная связь. Проблема была в том, что сервер запрашивал авторизацию всякий раз, тогда как клиент TFS делает это только при первом обращении. Файл хитрым образом проблему решает.

Новая проблема, папка Documents в Visual Studio заблокирована, хотя на веб-портале проекта она видна и можно добавлять туда файлы. Ладно, это не фатальная проблема. Главное — репозиторий исходников работает. Почти как на SVN.

Программная фабрика: дайте мне модель, и я сдвину Землю

Идея разрабатывать программы, минимизируя стадию кодирования на конкретных языках под заданные платформы, появилась достаточно давно. Прежде всего в связи с неудовлетворительной возможностью языков высокого уровня третьего поколения (3GL) описывать решаемые прикладные задачи в соответствующих терминах. За последнее время к этой причине добавилась ещё и поддержка независимости от целых платформ, ведь прогресс, как мы знаем, неотвратим, особенно «прогресс».

В управляемой моделями разработке¹ (УМР) и в программной фабрике² в частности наиболее интересной возможностью является генерация кода, скомпилировав который, можно сразу получить работающее приложение или его компоненты. Мы проектируем и сразу получаем нечто работающее, пусть даже на уровне прототипа. Уточняя модели, мы на каждом шаге имеем возможность видеть изменения в системе. Проектирование становится живым процессом без отрыва от разработки.

¹ В англоязычной среде Model Driven Architecture (MDA) и Model Driven Development (MDD).

² От англ. Software Factory.

Историю управляемой моделями архитектуры и разработки, обобщаемой под термином УМИ — управляемой моделями инженерии¹, можно вести с 1970-х годов. Именно тогда появились первые языки спецификации требований к программам и целые стандарты, типа упоминавшегося IDEF, включающего в себя ряд языков и нотаций. Однако реальная и доступная многим пользователям автоматизация моделирования появилась лишь вместе с персональными компьютерами. Не случайно форматы IDEF-диаграмм исторически сохранили рамки с ячейками информации, столь необходимыми при бумажной технологии анализа проектирования.

Первые инструменты CASE, выросшие из редакторов графических примитивов, были представлены в 1980-х годах, а одним из пионеров был небезызвестный Эдвард Йордон, соавтор, в компании с Томом ДеМарко, популярной методологии SADT². В начале 1990-х годов наблюдалось возникновение множества языков, нотаций, подходов к анализу и проектированию и, как следствие, пик многочисленных CASE-инструментов, их реализующих.

У программистов «от сохи» отношение к CASE, как правило, негативное на уровне «я не верю, что какие-то картинки генерируют код лучше написанного руками». В таком отношении есть своя сермяжная правда, действительно, экскаватор, в отличие от мужика с лопатой, может вырыть не всякую яму. Доказывать обратное — бесполезная потеря времени.

У более продвинутых программистов, имевших опыт написания и сопровождения тысяч строк однотипного рутинного кода, претензии становятся обоснованными и касаются, как правило, следующих сторон применения CASE-средств:

- Если ручное написание кода принять за максимальную гибкость, то CASE может навязывать каркас, стиль кодирования и шаблоны генерации частей программ, ограничивающие не столько полёт фантазии программиста, сколько возможность тонкой настройки генерируемого кода. Неважно, что такая настройка не требуется в большинстве случаев, но если её нет, то менять придётся непосредственно сгенерированный код.

¹ От англ. Model Driven Engineering.

² От англ. Structured Analysis and Design Technique — методология структурного анализа и проектирования.

- CASE работает только в условиях дисциплины, когда ручные изменения генерируемого кода исключены или автоматизированы (пост-обработка). Как только программист залезает руками в код каркаса, модель оказывается рассинхронизированной по отношению к исходным текстам программ и процесс разваливается.

В качестве решения перечисленных проблем появились так называемые двусторонние CASE-инструменты (*two way tools*), позволяющие редактировать как модель, непосредственно видя изменения в коде, так и, наоборот, менять код с полу- или полностью автоматической синхронизацией модели. Зачастую, такой инструмент был интегрирован прямо в среду разработки.



Рис. 17. Двусторонний CASE-инструментарий ModelMaker имеет возможности работы как с моделями, так и непосредственно с кодом приложения

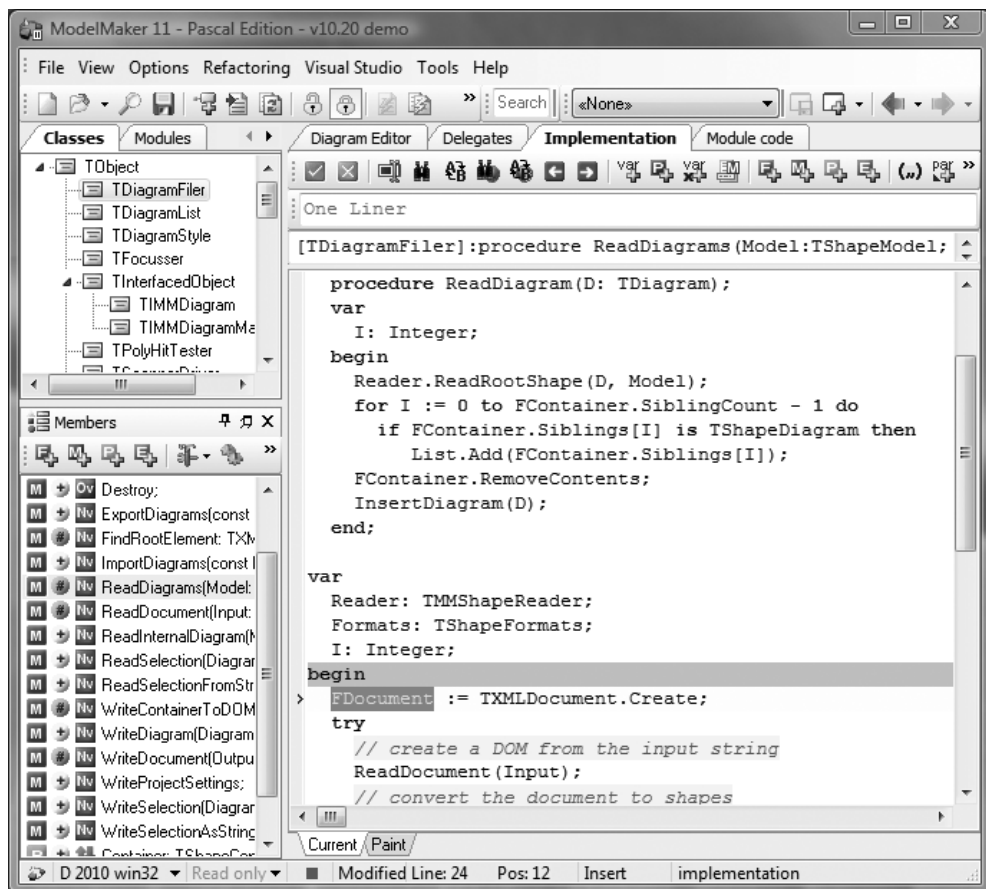


Рис. 18. Работа с кодом приложения в ModelMaker

Нетрудно заметить, что двунаправленный подход в CASE-инструментарии в большей степени является мощным средством автоматизации отдельных программистов, так как обладает рядом ограничений:

- как правило, инструмент привязан к языкам и платформам;
- технология не выходит за рамки разработки конкретных программ и подсистем. То есть слои системы и архитектура остаются за рамками процесса;

- коллективная работа над моделями одновременно с кодом практически невозможна: приходится делить модели на независимые части, например подсистемы, разрабатываемые одним программистом;
- для достижения нужного эффекта методика по-прежнему требует навыков моделирования как минимум на уровне диаграммы классов. В противном случае CASE оказывается лишь очередным инструментом рефакторинга.

Следующим шагом в развитии автоматизированных средств софстроения явилась программная фабрика — синтез подходов управляемой моделями разработки и архитектуры, генерирующий не только отдельные компоненты системы, но целые слои в соответствии с выбранной архитектурой и платформами.

На рынке уже имеется немало продуктов типа «*software factory*», если вы наберёте в поисковике эти ключевые слова, то получите множество ссылок на концепции и частные реализации. Например, неплохое руководство, хотя и привязанное к собственным средствам, составили в IBM[24]. Чтобы не утомлять вас текстами академического характера, в следующей главе я просто приведу пример одной фабрики под названием Genie Lamp (<http://genielamp.sourceforge.net>), применяемой непосредственно в различных моих проектах. Несмотря на то что подход УМР я использую с конца 1990-х годов, свести многие частные решения в несколько более общее удалось только за последние 2–3 года. Лень — двигатель прогресса, особенно когда надоедает переписывать генераторы кода и подстраивать относительно стандартные модели под частные требования.

Лампа, полная джиннов

Метафора системы достаточно проста: хочешь генерировать код компонента или слоя — попроси об этом соответствующего «джинна» в форме стандартного «заклинания». Джинны, как им и положено, живут в лампе.

Переходя к техническим терминам, программист описывает задачу в терминах логической модели, представляющей собой набор сущностей, их атрибутов, операций и связей между ними. Язык создан на основе XML, поэтому делать описания можно непосредственно руками в обычном текстовом редакторе.

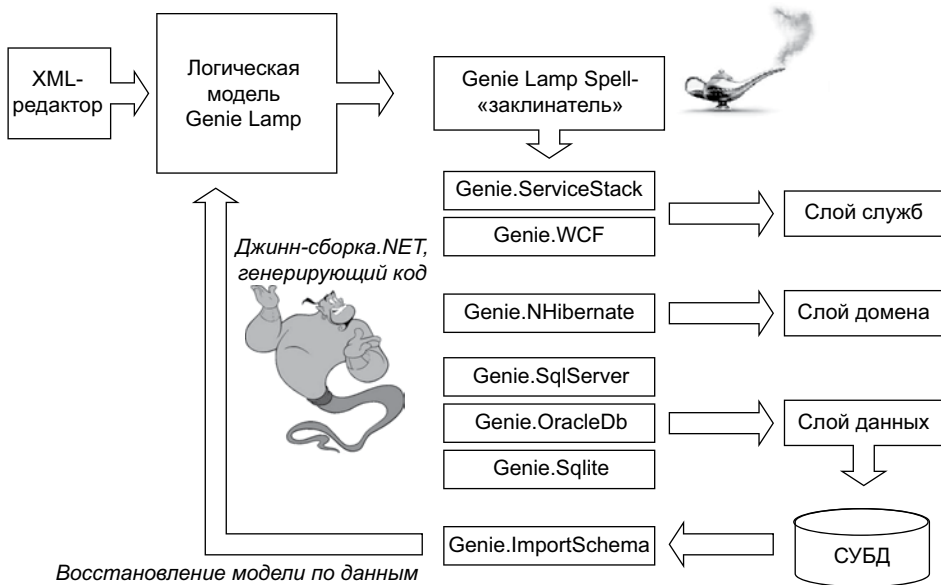


Рис. 19. Общая схема работы с «лампой» и «джиннами»

Модель в виде XML-файлов поступает на вход «заклинателю» — входящей в состав пакета консольной утилите. Производятся проверки непротиворечивости модели, выдающие ошибки либо предупреждения разной степени важности. Во время анализа модель также преобразуется во внутренний формат в виде множества объектов с открытыми интерфейсами доступа.

Если модель корректна, «заклинатель» начинает призывать «джиннов» сделать свою работу, передавая каждому на вход кроме самой модели ещё и разнообразные параметры, конфигурацию, касающуюся не только самих джиннов, но и, например, таких настроек, как правила именования в конкретном слое системы.

Обработав модель в соответствии с конфигурацией проекта, джинн выдаёт готовый к компиляции в среде разработки код. Для слоя хранения данных кроме генерации специфичных для СУБД SQL-скриптов производится их прогон на заданном сервере разработки.

В случаях, когда система уже существует и подлежит, например, переделке, можно восстановить модель из схемы базы данных. Конечно, даже теоретически такое восстановление не может быть полным из-за разницы в семантике, но большую часть рутинной работы оно выполняет. Проведя один раз импорт,

далее мы редактируем, структурируем модели и продолжаем работать только в обычном цикле изменений «через модель».

На что похожа логическая модель? Приведу пример описания из рабочего проекта, содержащего один пользовательский тип, один перечисляемый тип, две сущности и одну связь (отношение) между ними.

Пример модели в Genie Lamp

```
<Type name="TEntityId" baseType="int" />

<Enumeration name="Granularity">
  <Doc><Label lang="ru">Грануляция учётного периода</Label></Doc>
  <Item name="Day" value="0">
    <Doc><Label lang="ru">День</Label></Doc>
  </Item>
  <Item name="Month" value="1" default="true">
    <Doc><Label lang="ru">Месяц</Label></Doc>
  </Item>
  <Item name="Year" value="2">
    <Doc><Label lang="ru">Год</Label></Doc>
  </Item>
</Enumeration>

<Entity name="FiscalYear">
  <Doc><Label lang="ru">Финансовый год</Label></Doc>
  <Attribute name="Id" type="TEntityId" primaryid="true" autoincrement="true" />
  <Attribute name="Name" type="TCaption" uniqueid="true">
    <Doc><Label lang="ru">Обозначение года</Label></Doc>
  </Attribute>
  <Attribute name="Granularity" type="Granularity">
    <Doc><Label lang="ru">Грануляция периодов</Label></Doc>
  </Attribute>
  <Attribute name="FromDate" type="date">
    <Doc><Label lang="ru">Дата начала</Label></Doc>
  </Attribute>
  <Attribute name="ToDate" type="date">
    <Doc><Label lang="ru">Дата окончания</Label></Doc>
  </Attribute>
  <Attribute name="Closed" type="boolean" default="false">
    <Doc><Label lang="ru">Год закрыт?</Label></Doc>
  </Attribute>
  <Attribute name="GranularityName" type="string" persisted="false">
    <Doc>
      <Text lang="ru">Возвращает локализованное название грануляции</Text>
    </Doc>
  </Attribute>
  <Operation name="CreatePeriods" access="public">
    <Doc>
      <Text lang="ru">
```



```

        Создает периоды финансового года
        между датами начала и окончания
        в соответствии с грануляцией. Например, для фин.года,
        совпадающего с календарным, и помесечной грануляцией
        будут созданы 12 месячных периодов
    </Text>
</Doc>
<Returns type="void"/>
</Operation>
<Operation name="FindPeriodIdByDate" access="public">
    <Doc>
        <Text lang="ru">
            Возвращает ID периода по заданной дате, "0" если не найден
        </Text>
    </Doc>
    <Param name="periodDate" type="datetime"/>
    <Returns type="TEntityId"/>
</Operation>
<Operation name="DeleteCascade" access="public">
    <Returns type="void"/>
</Operation>
</Entity>

<Entity name="Period">
    <Doc><Label lang="ru">Учётный период</Label></Doc>
    <Attribute name="Id" type="TEntityId" primaryid="true" autoincrement="true" />
    <UniqueId>
        <Attribute name="FiscalYearId" type="TEntityId">
            <Doc><Label lang="ru">ID финансового года</Label></Doc>
        </Attribute>
        <Attribute name="FromDate" type="date">
            <Doc><Label lang="ru">Дата начала</Label></Doc>
        </Attribute>
    </UniqueId>
    <UniqueId>
        <OnAttribute name="FiscalYearId"/>
        <Attribute name="PeriodNumber" type="smallint">
            <Doc><Label lang="ru">Номер периода</Label></Doc>
        </Attribute>
    </UniqueId>
    <Attribute name="ToDate" type="date">
        <Doc><Label lang="ru">Дата окончания</Label></Doc>
    </Attribute>
</Entity>

<Relation entity="Period" name="FiscalYear"
    entity2="FiscalYear" name2="Periods"
    cardinality="M:1">
    <AttributeMatch attribute="FiscalYearId" attribute2="Id" />
</Relation>

```

Теперь необходимо задать конфигурацию в описании проекта. Предположим, что мы хотим создать 3-звенное приложение со следующими логическими слоями:

- слои хранения будут развёрнуты на SQL Server или Oracle;
- слой домена под управлением NHibernate;
- слой веб-служб на базе ServiceStack (вместо WCF, имеющего под Mono/Linux ограничения).

Пример конфигурации проекта в Genie Lamp

```
<!-- Включаем файл(ы) модели в проект -->
<ImportModel fileName="MyModel.xml" />

<!-- Будем использовать джинна SQL Server -->
<Genie name="SqlServer"
  type="GenieLamp.Genies.SqlServer.SqlServerGenie"
  assembly="GenieLamp.Genies.SqlServer"
  active="false"
  outDir="%PROJECT_DIR%/../SQL/SqlServer-%TARGET_VERSION%"
  outFileName="%PROJECT_NAME%.sql"
  updateDatabase="true"
  targetVersion="2008">
  <Param name="Database.Create" value="false" />
  ... Другие параметры "заклинания"
</Genie>

<!-- Будем использовать джинна Oracle -->
<Genie name="OracleDb"
  type="GenieLamp.Genies.Oracle.OracleGenie"
  assembly="GenieLamp.Genies.Oracle"
  active="true"
  outDir="%PROJECT_DIR%/../SQL/Oracle-%TARGET_VERSION%"
  outFileName="%PROJECT_NAME%.sql"
  outFileEncoding="iso-8859-1"
  updateDatabase="false"
  targetVersion="10g">
  <Param name="UniqueIndexConstraint" value="true" />
  ...
</Genie>

<!-- Будем использовать джинна NHibernate для генерации домена -->
<Genie name="NHibernate"
  type="GenieLamp.Genies.NHibernate.NHibernateGenie"
  assembly="GenieLamp.Genies.NHibernate"
  active="true"
  outDir="%PROJECT_DIR%/../Domain"
```

```

        outFileName="%PROJECT_NAME%.Domain.cs"
        targetVersion="*"
    <Param name="TargetAssemblyName" value="Company.Business.%PROJECT_NAME%.
Domain" />
</Genie>

<!-- Будем использовать первого джинна ServiceStack
для генерации интерфейсов к веб-службам -->
<Genie name="ServiceStack Services Interfaces"
type="GenieLamp.Genies.ServicesLayer.ServiceStack.ServicesInterfacesGenie"
assembly="GenieLamp.Genies.ServicesLayer"
active="true"
outDir="%PROJECT_DIR%/../Services.Interfaces"
targetVersion="*"
>
</Genie>

<!-- Будем использовать второго джинна ServiceStack
для генерации собственно веб-служб -->
<Genie name="ServiceStack Services"
type="GenieLamp.Genies.ServicesLayer.ServiceStack.ServicesGenie"
assembly="GenieLamp.Genies.ServicesLayer"
active="true"
outDir="%PROJECT_DIR%/../Services"
targetVersion="*"
>
</Genie>

<Configuration>
    <!-- Конфигурация слоя хранения данных -->
    <Layer name="Persistence">
        <NamingConvention style="uppercase" maxLength="30">
            <Param name="PrimaryKey.ColumnTemplate" value="NI%TABLE%" />
            <Param name="PrimaryKey.ConstraintTemplate" value="PK_%TABLE%" />
            ... Другие шаблоны именований
        </NamingConvention>
        <Param name="ForeignKey.CreateIndex" value="true" />
        <Param name="BooleanValues" value="YesNo"/>
    </Layer>

    <!-- Конфигурация слоя домена -->
    <Layer name="Domain">
        <Param name="BaseNamespace" value="Company.Business.%PROJECT_NAME%" />
    </Layer>

    <!-- Конфигурация слоя служб -->
    <Layer name="Services">
        <Param name="BaseNamespace" value="Company.Business.%PROJECT_NAME%" />
    </Layer>

    <!-- Шаблон "Реестр объектов" -->
    <Pattern name="Registry">
        <Param name="Schema" value="Core" />
        <Param name="PersistentSchema" value="CORE" />
        <Param name="RegistryEntity.Name" value="EntityRegistry" />

```

продолжение ⇨

```

    <Param name="TypesEntity.Name" value="EntityType" />
    <Param name="TypesEntity.PrimaryId.Type" value="smallint" />
    <Param name="PrimaryId.Type" value="bigint" />
</Pattern>

<!-- Шаблон "Версия состояния" для хранимых объектов -->
<Pattern name="StateVersion">
    <Param name="Attribute.Name" value="Version" />
    <Param name="Attribute.Type" value="int" />
</Pattern>

<!-- Шаблон "Аудит" для минимального отслеживания изменений -->
<Pattern name="Audit" />

<!-- Шаблон "Локализация" -->
<Pattern name="Localization" />

<!-- Шаблон "Безопасность" для веб-служб -->
<Pattern name="Security" />
</Configuration>

```

В описании конфигурации джиннов видно, что его основу составляет сборка, один из классов которой, реализующий интерфейс **IGenie**, является точкой входа. Каждый джинн имеет как общие для всех параметры, например каталог для выходных файлов, так и специфичные, передаваемые через тег **Param**, описываемые в документации.

За джиннами следуют конфигурации слоёв. Если для домена и служб можно пока ограничиться спецификацией базового пространства имён, то для слоя хранения, особенно при поддержке более чем одной СУБД, необходимо указать дополнительные ограничения вроде максимальной длины имён.

Заключительная часть конфигурации представляет собой описания шаблонов. Но не тех, о которых идёт речь в книжке «банды четырёх», а о шаблонах реализации типовых задач уровня ядра и системных служб:

- Например, **шаблон «Реестр объектов»** добавляет к системе возможность ведения централизованного реестра всех создаваемых объектов. Реализован он как соответствующий класс и таблица, ссылка на которые добавляется ко всем другим классам (некоторые классы можно исключить через параметры шаблона).
- **Шаблон «Версия состояния»** является встроенной в NHibernate возможностью отслеживания конфликтов в многопользовательской среде. Например, если два пользователя изменяют один и тот же объект, то последний из них, сохранивший объект, получит исключение,

оповещающее о том, что данные были изменены со времени последнего редактирования. Шаблон реализуется добавлением соответствующего атрибута номера версии ко всем классам.

- **Шаблон «Аудит»** в простейшем варианте является регистрацией для каждого хранимого объекта информации о времени его создания, последнем редактировании и авторе.
- **Шаблон «Локализация»** добавляет в генерируемый код возможность перевода сообщений в рамках технологии GNU gettext.
- Наконец, **шаблон «Безопасность»** в простейшем варианте ограничивает доступ к веб-службам через механизм аутентификации, логику которой необходимо реализовать в переопределяемом методе соответствующего класса. Например, обратиться к стороннему LDAP или непосредственно к базе данных с регистрационной информацией для проверки имени пользователя и хеша пароля.

Теперь, если запустить «заклинатель» с параметром файла конфигурации проекта и не будет обнаружено ошибок, на выходе мы получим инициализированные структуры баз данных и готовые к компиляции файлы. Рассмотрим их чуть подробнее.

Слой хранения (СУБД)

Джинны SQL Server и Oracle создадут нам в указанном каталоге подкаталоги, соответствующие целевой СУБД и её версии. В каждом подкаталоге находятся три SQL-скрипта, предназначенные, соответственно, для создания, обновления или удаления схемы БД.

```
SQL
├── Oracle-10g
│   ├── CRE_MyProject.sql
│   ├── DEL_MyProject.sql
│   └── UPD_MyProject.sql
└── SqlServer-2008
    ├── CRE_MyProject.sql
    ├── DEL_MyProject.sql
    └── UPD_MyProject.sql
```

Если посмотреть на созданные в СУБД структуры, то мы увидим, что из одной и той же модели логического уровня были созданы две реализации, различающиеся на физическом уровне. Например, используемые типы данных различаются. С другой стороны, необходимость поддержки слоев домена сразу двух СУБД приводит к тому, что вместо оптимального, но специфичного для SQL Server типа `bit` для поддержки булевых величин используется принятый в среде Oracle символьный тип.

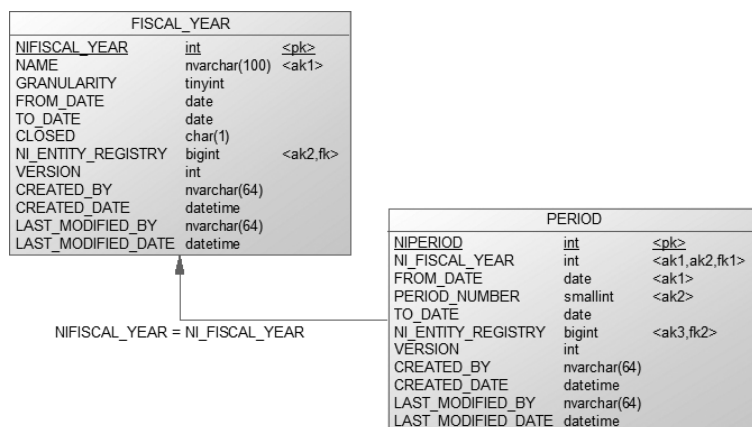


Рис. 20. Результат работы джинна SQL Server

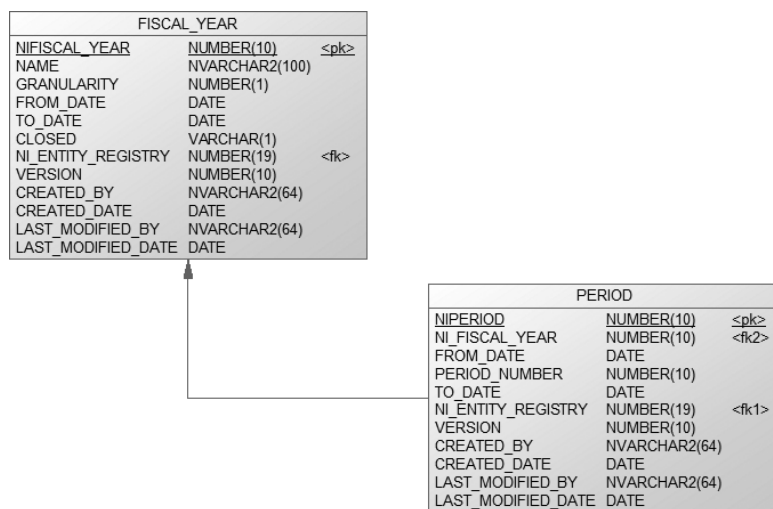


Рис. 21. Результат работы джинна Oracle

Слой домена (NHibernate)

Джинн NHibernate генерирует три C#-файла и один XML-файл проекции (маппинга) классов на структуры хранения.

Файл	Назначение
<Название>.Domain.cs	Классы домена согласно модели
<Название>.Domain.hbm.xml	Проекция классов на структуры для NHibernate
DomainSetup.cs	Инициализация данных объектов домена
DomainSupport.cs	Служебные классы и обработчики событий

Все классы домена являются расширяемыми (*partial*), что позволяет разработчику вынести специфичные не поддерживаемые моделью реализации свойств и методов классов в отдельные файлы.

Перечисляемый тип создаётся вместе с классами, позволяющими локализовать пользовательские названия его элементов, описанные в модели. По умолчанию будет использован язык модели, перевод необходимо осуществлять в технологии *gettext*.

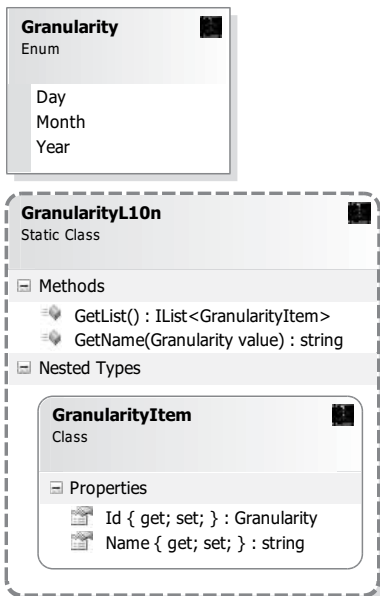


Рис. 22. Перечисляемый тип слоя домена и его локализация

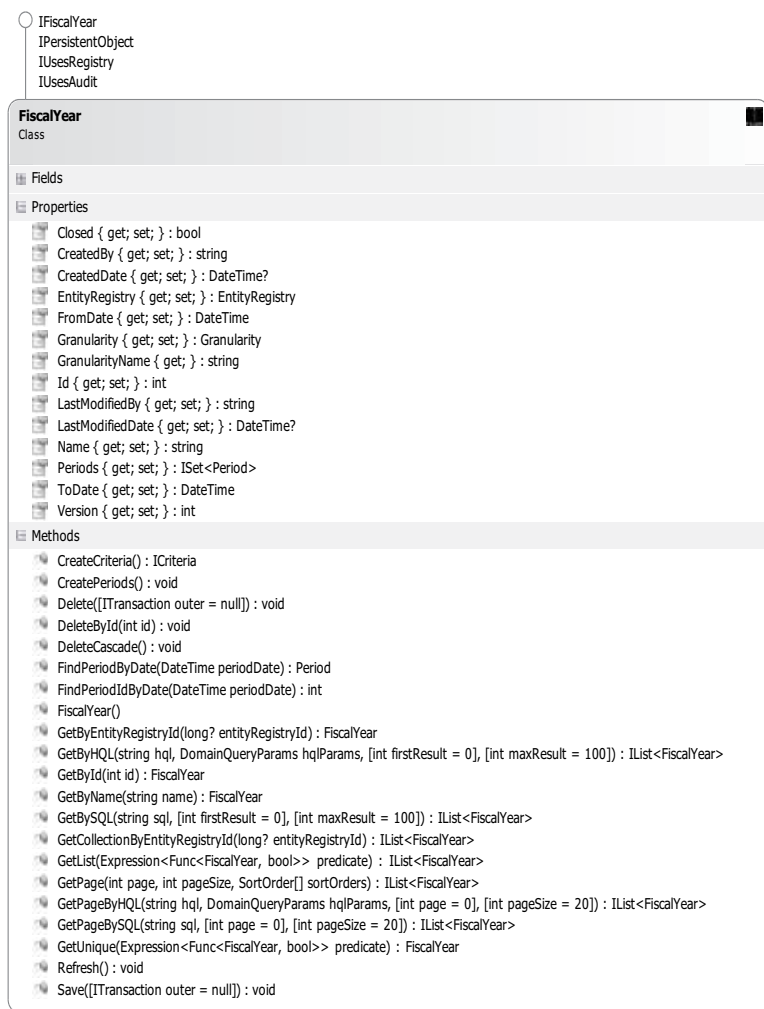


Рис. 23. Класс «Финансовый год» слоя домена

Для определяемых моделью сущностей генерируются классы, содержащие кроме соответствующих объявленным атрибутам свойств ещё и группу служебных методов для управления объектами, например, для сохранения или выборки по запросу на HQL или даже SQL. Такие методы часто используют служебные классы, объявленные в **DomainSupport.cs**.

Для объявленных в модели операций генерируется соответствующий метод интерфейса. Реализовать их нужно программисту в рамках расширения *partial*-класса.

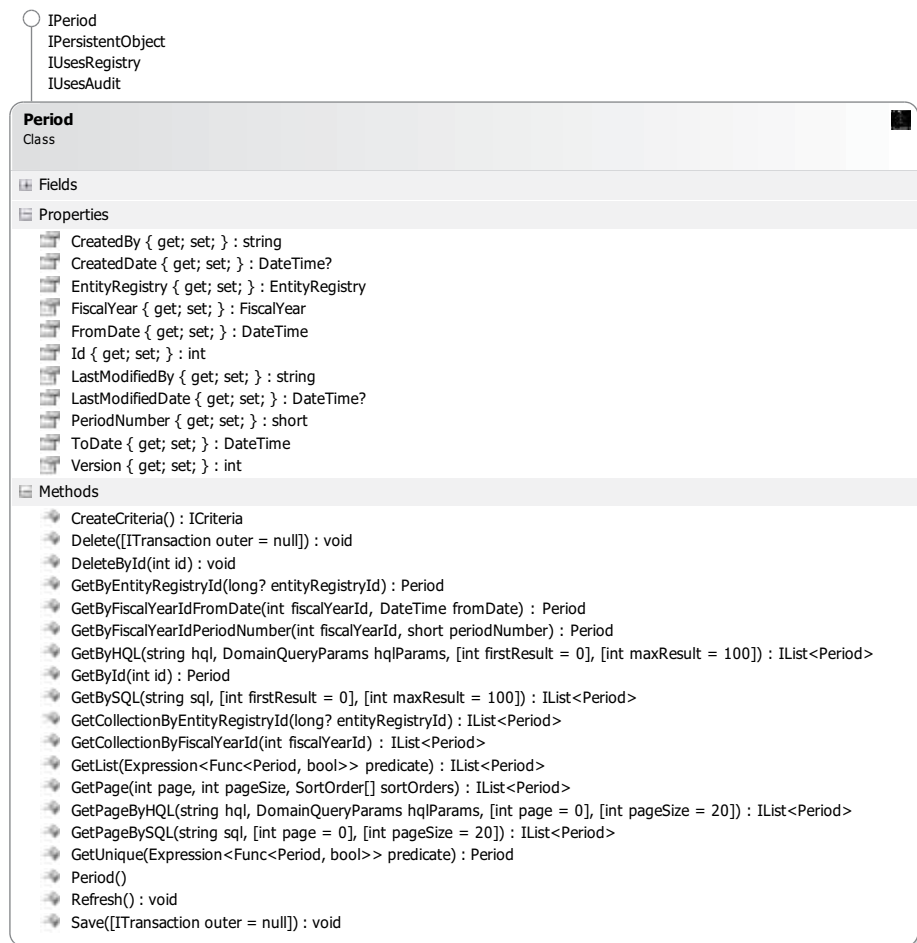


Рис. 24. Класс «Учётный период» слоя домена

Отображение классов на структуры РСУБД соответствует стратегии «одна таблица на подкласс без дублирования атрибутов предка», достаточно хорошей в большинстве случаев, но и она может быть изменена.

Фрагмент файла проекции для класса «Финансовый год»

```

<!--Engine.FiscalYear-->
<class name="Domain.Engine.FiscalYear" table="FISCAL_YEAR" schema="MYAPP">
  <id name="Id" access="property" column="NIFISCAL_YEAR">

```

продолжение ➞

```

    <generator class="native">
      <param name="sequence">MYAPP.SEQ_FISCAL_YEAR</param>
    </generator>
  </id>
  <version name="Version" column="VERSION" type="int" access="property" />
  <set name="Periods" table="MYAPP.PERIOD" inverse="true" lazy="true"
cascade="none">
    <key column="NI_FISCAL_YEAR" />
    <one-to-many class="Domain.Engine.Period" />
  </set>
  <many-to-one name="EntityRegistry" class="Domain.Core.EntityRegistry"
unique="true">
    <column name="NI_ENTITY_REGISTRY" not-null="false" />
  </many-to-one>
  <property name="Name" access="property">
    <column name="NAME" not-null="false" />
  </property>
  <property name="Granularity" access="property">
    <column name="GRANULARITY" not-null="true" />
  </property>
  <property name="FromDate" access="property">
    <column name="FROM_DATE" not-null="true" />
  </property>
  <property name="ToDate" access="property">
    <column name="TO_DATE" not-null="true" />
  </property>
  <property name="Closed" access="property" type="YesNo">
    <column name="CLOSED" not-null="true" />
  </property>
  <property name="CreatedBy" access="property">
    <column name="CREATED_BY" not-null="false" />
  </property>
  <property name="CreatedDate" access="property" type="timestamp">
    <column name="CREATED_DATE" not-null="false" />
  </property>
  <property name="LastModifiedBy" access="property">
    <column name="LAST_MODIFIED_BY" not-null="false" />
  </property>
  <property name="LastModifiedDate" access="property" type="timestamp">
    <column name="LAST_MODIFIED_DATE" not-null="false" />
  </property>
</class>

```

Слой веб-служб и интерфейсов доступа (ServiceStack)

Генерируемые для слоя веб-служб C#-файлы предназначены для создания двух сборок: собственно служб и интерфейсов к ним, используемых клиентами.

Сборка	Файл	Назначение
Службы	DomainServices.cs	Классы, реализующие веб-службы доступа к домену
	DomainServicesHost.cs	Класс для автономного сервиса, например, консольного приложения, unix-демона или сервиса Windows
Интерфейсы	ServicesInterfaces.cs	Классы доступа к сервисам, контракты
	DomainDTO.cs	Классы DTO для взаимодействия с доменом
	DomainDTOAdapters.cs	Адаптеры над классами DTO, упрощающие их использование в прикладном коде программ-клиентов

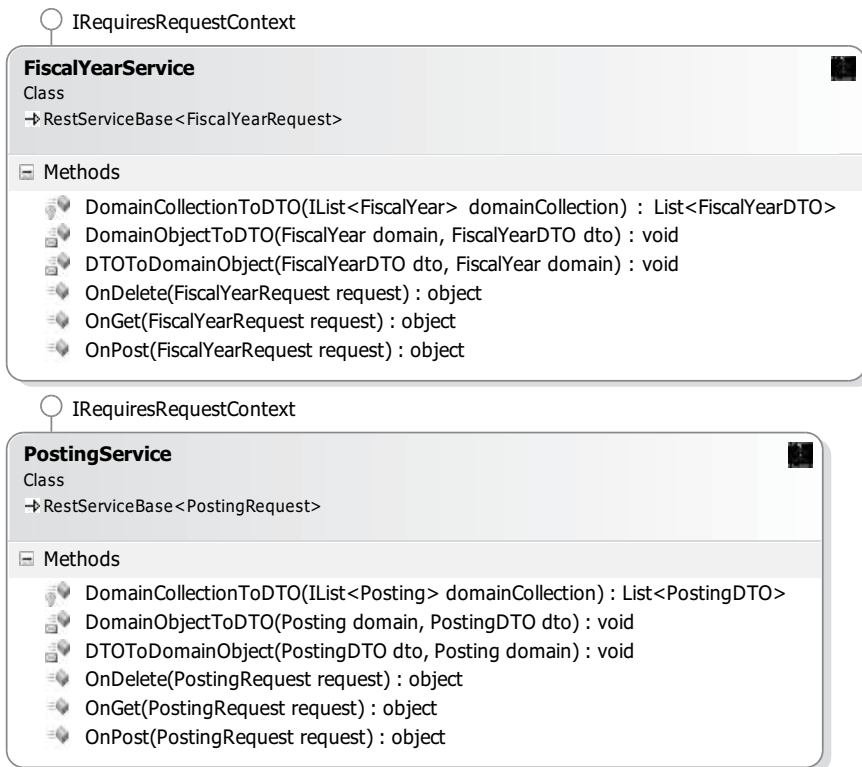


Рис. 25. Классы, реализующие службы доступа к объектам домена

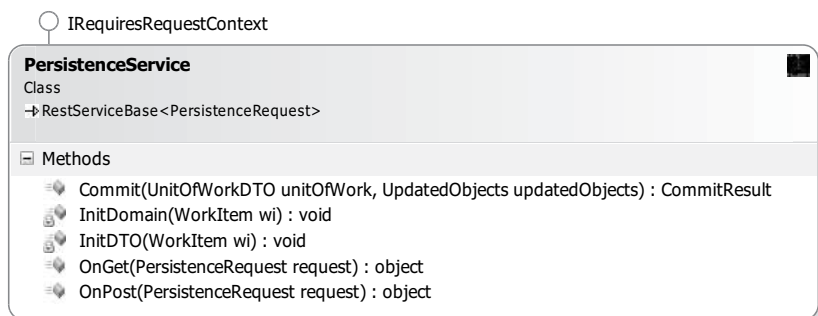


Рис. 26. Класс службы сохранения объектов

Интерфейсы доступа к службам также содержат описания перечислимых типов с локализацией, классы DTO для передачи состояния между программой-клиентом и доменом, классы для непосредственного доступа к вызовам служб.

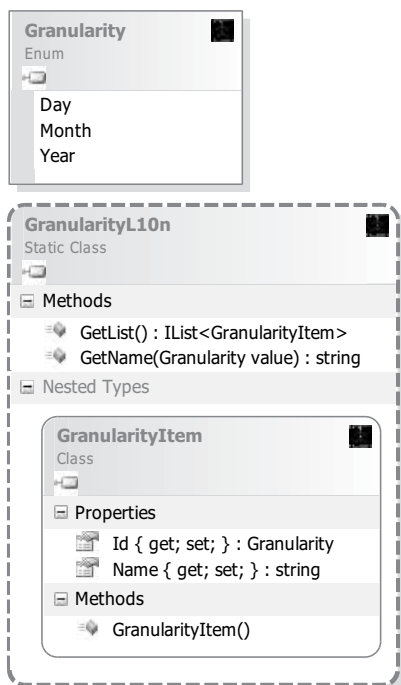


Рис. 27. Перечисляемый тип слоя веб-служб

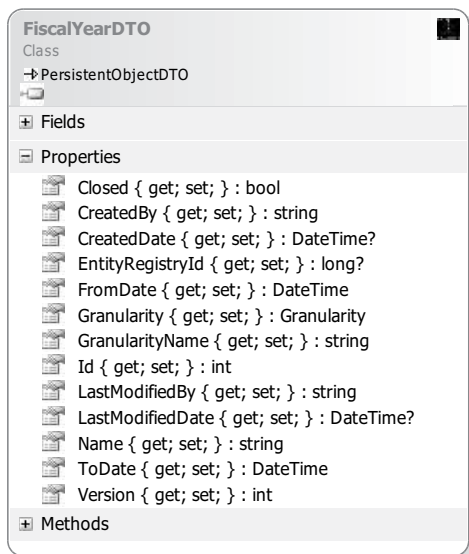
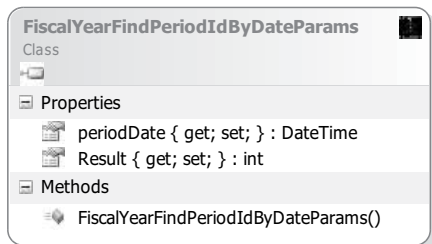
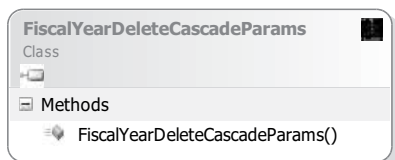
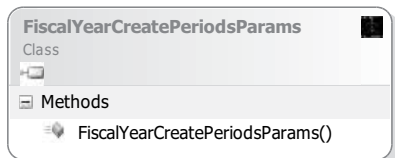


Рис. 28. Классы вызова специфицированных методов

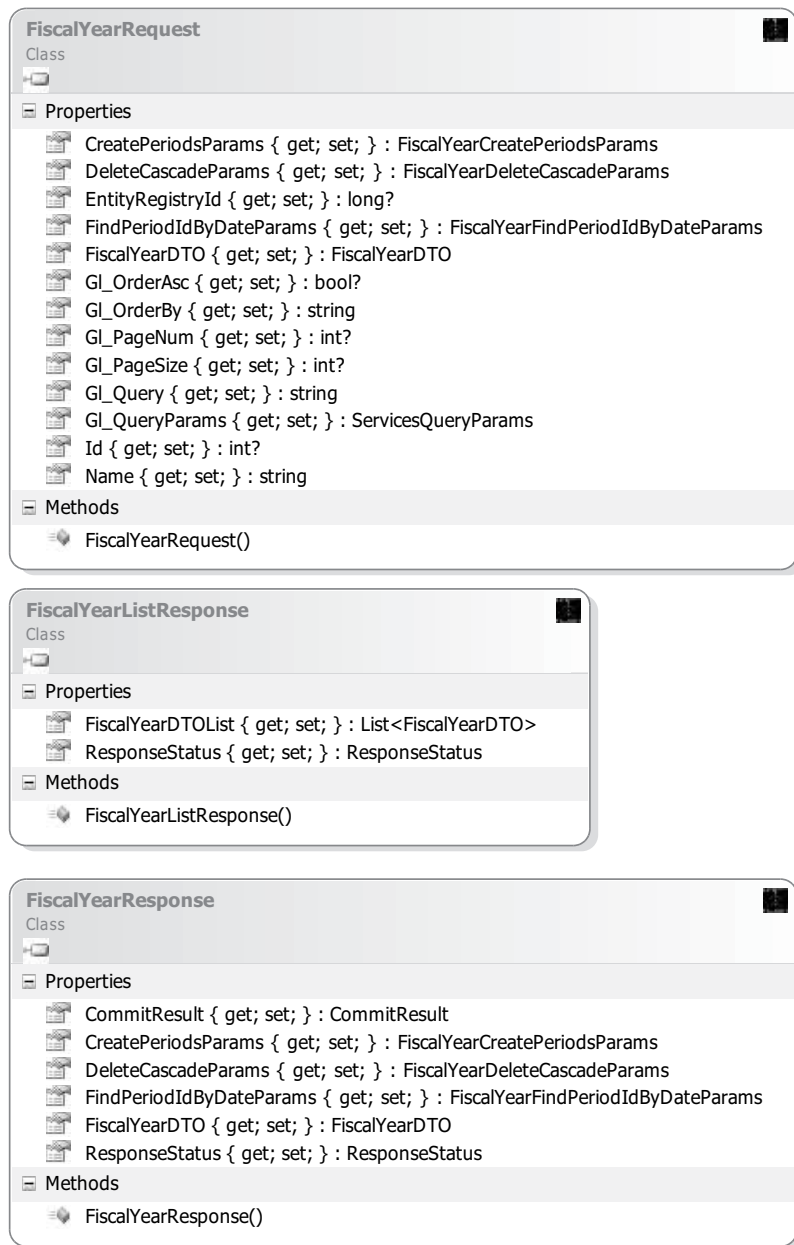


Рис. 29. Классы вызова веб-служб, касающихся «финансового года»



Рис. 30. Класс адаптера для работы с объектом «Финансовый год»

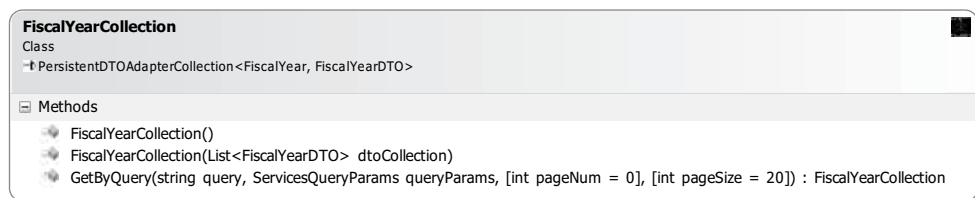


Рис. 31. Класс адаптера для работы с коллекцией объектов «Финансовый год»

Работать с DTO и коллекциями не слишком комфортно, проявляется много ненужных деталей. Но если обернуть операции с DTO адаптерами, то код становится гораздо более читаемым и коротким.

Пример работы с DTO

```
CurrencyDTO curr1 = new CurrencyDTO();
curr1.Code = "RUR";
curr1.Name = "Currency 1";
UnitOfWorkDTO uow = new UnitOfWorkDTO();
uow.Save(curr1);

PersistenceRequest prq1 = new PersistenceRequest();
prq1.UnitOfWork = uow;
PersistenceResponse prr1 = client.Post<PersistenceResponse>("/Persistence", prq1);
Assert.IsFalse(prr1.CommitResult.HasError, prr1.CommitResult.Message);
```

Пример работы с адаптерами

```
Currency curr1 = new Currency();
curr1.Code = "RUR";
curr1.Name = "Currency 1";
CommitResult cr1 = curr1.Save();
Assert.IsFalse(cr1.HasError, cr1.Message);
```

Программа-клиент

В рамках простейшего WinForms-приложения создадим форму, содержащую сетки отображения финансовых годов и их периодов. Не вдаваясь в технику разработки приложений этого типа, просто приведу фрагменты кода, запрашивающие у служб коллекции соответствующих типов.

Извлечение списка финансовых годов, отфильтрованного по названию

```
FiscalYearCollection years = FiscalYearCollection.GetByQuery(
    "from FiscalYear where Name like :name order by Name ",
    new ServicesQueryParams()
    .AddParam("name", txtYearName.Text)
);
dgvYears.DataSource = years;
```

Извлечение списка учётных периодов заданного года

```
PeriodCollection periods = PeriodCollection.GetByQuery(
    "from Period where FiscalYear.Id = :yearId order by FromDate",
    new ServicesQueryParams()
    .AddParam("yearId", CurrentYear.Id),
    0, 1000);
dgvPeriods.DataSource = periods;
```


Запускаем клиентское приложение, предварительно запустив сервер веб-служб, и видим на экране примерно такую картинку, как на рис. 32.

The screenshot shows a window titled "Финансовые годы и периоды" (Financial years and periods). Inside, there is a section for adding a financial year with a text input "Название финансового года" (Financial year name) containing "Test%", a "Просмотр" (View) button, and buttons for "Добавить" (Add), "Удалить" (Delete), and "Сохранить" (Save). Below this is a table of financial periods.

	Название	Дата начала	Дата окончания	Закрыт?
▶	Test period (created 03/10/...	01/01/2009	01/01/2010	<input type="checkbox"/>

Below the table is a "Показать периоды" (Show periods) button and another table of accounting periods.

	Номер	Дата начала	Дата окончания
▶	1	01/01/2009	01/02/2009
	2	01/02/2009	01/03/2009
	3	01/03/2009	01/04/2009
	4	01/04/2009	01/05/2009
	5	01/05/2009	01/06/2009
	6	01/06/2009	01/07/2009
	7	01/07/2009	01/08/2009
	8	01/08/2009	01/09/2009
	9	01/09/2009	01/10/2009
	10	01/10/2009	01/11/2009

To the right of the second table is a "Создать периоды" (Create periods) button.

Рис. 32. Форма отображения финансовых годов и учётных периодов

Остановиться и оглянуться

Рассмотренная выше подсистема состоит из минимального набора слоёв трёхзвенной архитектуры на основе веб-служб. Тем не менее даже в таком минимальном варианте обилие деталей, промежуточных и служебных классов, проекций и преобразований должно дать представление о проблеме сложности современного состояния софтверного строения.

Одним из способов обхода этой проблемы является описанная технология программной фабрики, несомненно далёкая от совершенства и ограниченная в наборе целевых платформ.

Какова же эффективность?

Если рассмотреть метрики относительно небольшого проекта, то 40 прикладных сущностей в модели, состоящей примерно из 600 строк XML-описаний, порождают:

- около 3 тысяч строк SQL-скриптов для каждой из целевых СУБД;
- порядка 10 тысяч строк домена;
- 1200 строк XML для проекций классов на реляционные структуры (таблицы);
- около 17 тысяч строк веб-служб и интерфейсов.

Таким образом, соотношение числа строк мета-кода описания модели к коду его реализации на конкретных архитектурах и платформах составляет около 600 к 30 тысячам или **1 к 50**.

Это означает, что оснащённый средствами автоматизации программист с навыками моделирования на этапе разработки рутинного и специфичного для платформ/архитектур кода производитель примерно так же, как и его 50 коллег, не владеющих технологией генерации кода по моделям. Любое внесение изменений в модель тут же приводит в соответствие все генерируемые слои системы, что ещё более увеличивает разрыв по сравнению с ручными модификациями. Наконец, для генерируемого кода не нужны тесты. Производительность возрастает ещё как минимум вдвое.

Даже если принять во внимание, что доля рутинного и прочего инфраструктурного кода по отношению к прикладному, то есть решающему собственно задачи конечных пользователей, снижается с масштабом системы, есть о чём поразмыслить в спокойной обстановке.

Cherchez le bug, или Программирование по-французски

Этот рассказ я написал более 10 лет назад, летом 2001 года, в поездках на пригородном поезде между Парижем и Moulin-Galant, где размещался филиал IBM, и поначалу сомневался в необходимости его включения в книгу. Но, просмотрев старый текст, с некоторым удивлением я обнаружил, что если за-

менить аббревиатуры в названиях технологий на «новые и прогрессивные», то суть повествования останется прежней. Изменится ли что-нибудь ещё через 10 лет, покажет время.

Хаос наступает внезапно

Что такое «баг» знают, наверное, все программисты. Кто не знает, поясню: «баг» (от англ. bug — жучок) — это «жучок-вредитель» в программе, то есть ошибка, аномалия, сбой. По-французски это интернациональное словечко произносится примерно как «бёг», но буква «ё» произносится ближе к звуку «о». Особенностью французской, как, собственно, и любой другой программной индустрии — громкое слово, несколько облагораживающее нашу всемирную «багодельню», является одно из национальных состояний французской души, характеризующее этакого сангвиника после распитого с друзьями бокала вина, галантного и совсем не торопящегося жить. Разумеется, сангвиник считает, что все продукты его труда велики и прекрасны, как вид на ночной Париж с Монмартрского холма или пыльный гобелен над кроватью Наполеона в замке Фонтенбло.

Первый опыт пришёл ко мне уже через пару недель после начала работы на новом месте. Не считая себя экспертом по программированию на C++, увидев исходные тексты, я ощутил мягко говоря, некоторое беспокойство. Столкнулся я с этими текстами не сразу, так, прежде в конторе трудились двое русских программистов, один из которых по разным причинам покинул фирму, а с другим, Димой, мы работали немногим более года вплоть до её ликвидации.

Итак, некоторое время я находился в счастливом неведении относительно общего состояния дел. Прежде всего пришлось оценить качество кода. То, что единые соглашения отсутствовали, а Java-подобный стиль не очень хорош в программах на C++ не являлось большим недостатком — лучше такой стиль, чем никакой, тем более что кода на Java в системе было много. Прежние авторы, видимо, не смогли выразить себя в объектно-ориентированном подходе, а до структурного программирования и функциональной декомпозиции не опустились. Если первое не сразу бросалось в глаза наличием достаточного числа классов с пустыми конструкторами, то второе резало глаз обилием возвратов из разных мест одной функции и очень похожими кусками кода в одноимённых перегруженных функциях, написанных методом копирования текста через буфер обмена, «copy-paste». При виде некоторых фрагментов кроме чисто рус-

ских эмоциональных выражений из меня периодически вылезало французское «что за бордель», вызвавшее похвалу моего коллеги, отмечавшего, что я делаю успехи в освоении языка.

В жизни каждого мало-мальски сложного программного продукта есть стадия, когда система проходит некий порог увеличения сложности, за которым наступает состояние, которое я называю «самостоятельной жизнью». Это ещё далеко не полный хаос, но уже давно и далеко не порядок. Все попытки как-то организовать процесс разработки программ, всяческие методологии, применение парадигмы конвейера, стандарты и административные меры худо-бедно, но помогают оттянуть этот критический порог на некоторое время. В идеале — до того момента, когда развитие системы останавливается и она, побыв некоторое время в стабильном состоянии, потихоньку умирает. Одна из проблем организации промышленного производства программного обеспечения состоит в отсутствии каких-либо формальных описаний деятельности программиста. Можно определить в технологической карте, как работает сварщик или каменщик, но как пишет программу программист зачастую не знает и он сам. До художника, конечно, далеко не все дотягивают, а вот с деятельностью рядового журналиста «независимой» газеты непосвящённому в софстроение человеку сравнивать вполне можно. Этаким ядрёный сплав ремесла, некоего богемного искусства, со вкраплениями науки, вперемешку с халтурой, шашкой и постоянным авралом. Попытки же принудить программиста делать однотипные операции противоречат самой цели существования программного обеспечения как самого гибкого из существующих средств автоматизации рутинных процессов и потому изначально обречены на неудачу.

Вернемся к нашим «жучкам». Система, с которой я начал работать, уже успешно прошла свой порог несколько месяцев назад и жила полнокровной, отдельной от авторов жизнью. Определить, что критический порог пройден, несложно, для этого у меня есть один простой признак: «Ты изменил чуть-чуть программу в одном месте, но вдруг появилась ошибка в другом, причём даже автор этого самого места не может сразу понять, в чем же дело»¹. Существует и второй признак, не менее практичный: «Смотришь на чужой текст программы, и тебе не вполне понятен смысл одной его половины, а вторую половину тебе хочется тут же полностью переписать».

Поскольку я определил, что налицо сразу два признака, то, кроме как включиться в борьбу за продление жизни системы, уже ничего не оставалось.

¹ Более формальное определение — «хрупкий дизайн».

При самых удачных раскладах такая борьба может длиться годами. Например, в таком богатом и «баратом» учреждении, как Microsoft, несколько лет боролись за жизнь Windows 95, выпустили даже Windows 98, но в результате все-таки сил осталось только на Windows 2000. Для неспециалистов это может быть неочевидно, поэтому поверьте мне на слово, что эти системы совершенно разные, как дельфин и русалка. Второй способ: прекратить текущую разработку программы и начать новую, используя опыт предыдущего прототипа. На это кроме обычного мужества признания ошибок и риска полететь с должности начальника требуется ещё и немало денег. Третий способ — выдать желаемое за действительное, побольше маркетинга, шуму, «подцепить» несколько заказчиков и на их деньги попытаться всё-таки перейти ко второму или первому способу. Такой трюк может сработать, если заказчику честно предлагают за какие-то вкусные для него коврижки побыть немного «подопытным кроликом», на котором система вскоре «должна заработать».

Кстати, если программист говорит вам, что в данном месте программа «должна работать», это значит, что с очень большой вероятностью она не работает, а он её просто не проверял в этом месте. Если же программист уже после обнаружения ошибки говорит: «А у меня она в этом месте работает...», лучше сразу его уволить, чтобы не мучился. Это проза жизни, также относящаяся к коллекции моих практик.

Контора пошла по третьему пути с прицелом на последующий переход к первому. Одно из самых скромных маркетинговых утверждений на рекламном проспекте гласило: «Наша система работает под Windows и под UNIX, она может взаимодействовать с любыми базами данных». Далее шёл список названий СУБД из первой десятки, в одном названии была ошибка. Реально же, уже к моменту, когда порог сложности был перейдён, имелась только версия программных модулей с постоянным номером, «текущая» под Windows NT и отстающая от нее на пару недель под Linux Mandrake, а из баз данных использовался Microsoft Access. Заказчики ходили косяками, поэтому шанс найти авантюриста увеличивался от недели к неделе. Но он всё не находился. И только благодаря тому, что у пары конкурирующих контор дела обстояли ещё хуже (это было для меня просто потрясающей новостью!), некая достаточно крупная фирма после небольшого конкурса решила на опытное внедрение. Её примеру последовала и вторая. Тут-то и начались настоящие приключения.

Что-то с памятью моей стало

Отступив от основной темы, опишу в общих чертах структуру конторы, в которой мы работали. Основателей у фирмы двое: генеральная директриса Софи — экспрессивная француженка, незамужняя, уже в годах, и технический директор Блез — почти наш ровесник, но, по-видимому, реально взявшийся за свой первый проект. Тут ведь всё неспешно происходит, студенты учатся, а не работают, поэтому такая ситуация для специалиста к тридцати годам нормальна. Софи заведует общими вопросами и руководит подразделением маркетинга, то есть создаёт полезный фон или, наоборот, шумовые помехи в зависимости от ситуации и квалификации представителей заказчика. Блез, соответственно, должен заниматься созданием продукта. Изначально продукт делался силами нескольких человек в течение полугода, включая самого техдиректора, после чего кое-кто ушёл, сам Блез понял, что программировать он уже не успевает, а некоторые оставшиеся просто откровенно не умеют. В результате появились двое русских программистов, на которых легла вся системная разработка и поддержка последнего уровня. Из тех, кто программировать не умеет, была составлена группа прикладных программистов и некое подобие группы поддержки всего процесса в виде администрирования репозитория исходного кода, создания резервных копий и дистрибутивов. В задачу прикладного программиста должно было входить быстрое создание веб-приложения для заказчика на основе собственно продукта. Итого набралось человек 15, включая директоров и весёлый шумный маркетинг.

Центральным звеном системы является модуль с типичным названием «ядро» (*kernel*). Многострадальное ядро, несмотря на месяцы групповых измывательств со стороны коллектива программистов с меняющимся составом, каким-то чудом все-таки компилировалось и запускалось, хотя нам приходилось прибегать к помощи «лома и такой-то матери».

Заказчики, конечно, и не подозревали о том, что месяцев десять назад ядро состояло из кучки «ядрышек», которые просто слили в один кусок, когда на первых запусках время ожидания ответа не устроило даже самих авторов. Однако после результатов такого слияния бдительность была потеряна надолго, хотя ничто не мешало параллельно с разработкой делать хотя бы простые тесты и прогоны. В результате после первых дней работы системы в условиях опытной эксплуатации наш технический директор стал ещё более нервным, чем раньше, особенно когда Дима или я заводили разговор о том, что «вообще-то это все не будет работать» и надо не перекраивать испорченный костюмчик, а шить новый, хотя бы по старым выкройкам.

Нервность Блеза раньше выражалась в том, что он просто частенько прохаживался по нашей комнате и раз в неделю, вдохновившись беседами с потенциальными клиентами, выдавал очередную гениальную идею о развитии продукта, иногда противоречащую идее предыдущей недели, но всегда очень расплывчатую и вызывающую необходимость нам с Димой садиться и писать некое подобие спецификации. Созданная спецификация Блезом практически никогда не менялась и периодически вовсе игнорировалась, поскольку это означало необходимость отвечать за свой же «базар» в уже конкретном формальном виде, но нам она была нужна для самоконтроля и взаимодействия, к тому же ее мы выдавали ещё и как заменитель инструкции для прикладного программиста.

Усиление нервозности выражалось в хождении по комнате, причём с гораздо большей скоростью, теперь уже практически всё время в перерывах между поездками к потенциальным заказчикам и встречами с ними в офисе. К тому же теперь хождение часто сопровождалось разговорами по телефону. Блез брал аппарат в правую руку, трубку в левую и объяснял заказчику недокументированные тонкости души продукта, попутно периодически задевая шнур и роняя телефон. На самом деле я очень не люблю, когда в комнате, где люди работают большее время дня в сидячем положении и относительной тишине, начинает кто-то мельтешить перед глазами и шуметь. Думаю, многие будут со мной солидарны. Отвлекает.

После нескольких дней падений блезовского телефона выяснилось, что ядро с немереным аппетитом кушает оперативную память сервера и никак не хочет отдавать ее обратно даже после отключения всех пользователей. Утечки памяти составили около одного гигабайта (!) в сутки. Мы принялись за тесты, которые по-хорошему надо было бы начинать ещё первым авторам месяцев десять назад, и на напоминания о необходимости которых уже с нашей стороны ответов не следовало. Тесты на небольшом макете показывали примерно ту же картину, что и у заказчика, оставалось со спокойным сердцем сесть и разобраться в причинах. Однако времени, как выяснилось, уже не было. Прибежавшая в восемь часов вечера Софи застала нас с Димой уже практически в дверях и сообщила, что все плохо, надо срочно что-то делать, иначе контора может просто накрыться.

В тот день мы просидели до трех часов ночи, на следующий день ещё до часу, пришлось выйти и в воскресенье. Таким образом, три дня к отпуску я заработал, сам того не желая. Но результаты были не очень хорошими: удалось только ликвидировать утечки памяти, но остались проблемы взаимных блокировок,

которые появлялись изредка и в совершенно разных ситуациях. Если вспомнить, на что похожи исходные тексты программ, то поиск таких ситуаций был на порядок сложнее поиска иголки в стоге сена.

Три дня в IBM

У второго заказчика был сервер от всемирно известной фирмы IBM, под управлением операционной системы AIX — IBM-овского варианта UNIX. Поскольку нашим маркетингом заявлялось, что система будет работать чуть ли не везде, то по русской поговорке «назвался груздем...» следовало лезть в этот самый кузов. Блез договорился с филиалом IBM о том, что в их демонстрационно-учебном зале несколько дней поработают программисты. Дима, как единственный из нас двоих знакомый с Linux, ответственный за компиляцию и сведение версии ядра под него, был настроен оптимистично и поехал собирать систему под AIX. Я остался в офисе делать сервис под Windows NT, который управлял бы запуском нашей системы в рабочем режиме.

Однако прошла неделя, а от Димы приходили только редкие послания с описаниями новых проблем. Например, компилятор C++ от IBM не переваривал некоторые конструкции, которые заявлялись как стандартные, но не оказывались таковыми на практике. Поскольку система состоит из многих собственных компонентов на C++ и Java, а также нескольких библиотек сторонних разработчиков, то проблемы нарастали как снежный ком, причём всплыла и старая, казалось бы, ликвидированная беда с утечкой памяти, появились уже стабильные признаки блокировки процесса. В итоге Блез решил отправить меня на подмогу Диме, поскольку мы периодически с ним весьма эффективно практиковали парное программирование — один из полезных методов для работы в экстремальных условиях.

Филиал IBM находился в 25 километрах к юго-востоку от Парижа, недалеко от железнодорожной станции с названием Moulin-Galant. Дорога туда не ближняя и занимала около полутора часов от дома до входа в офис, располагавшийся в помещениях завода по производству полупроводников. Я мысленно пожалел бедного коллегу, который уже вторую неделю каждый день вставал в 7 утра и приезжал домой в 10 часов вечера. От станции надо пройти пешком 15–20 минут, но, когда мы шли, то кроме ещё одного человека ни одной живой души на расстоянии нескольких километров не было, только проезжали по узкому шоссе машины. Вдоль дороги тесными рядами стояли домики и мало-

этажные постройки — типичный европейский пейзаж. За все три дня ни одного человека во внутренних садах этих домиков я так и не увидел.

Первый день мы начали с экспериментов. Сделали тестовое приложение, которое работало со сторонней библиотекой, и через пару часов убедились, что память расходуется именно в ней. Заодно, даже не помню точно, как, выяснилось, что устойчивая ошибка, приводящая к краху ядра, была обусловлена настройками памяти на уровне операционной системы. Консультант, дядька лет пятидесяти, исправил нам эти параметры и пожелал дальнейших успехов. Кстати, у них там вся команда консультантов и менеджеров состояла из мужчин предпенсионного возраста — факт отрадный, если вспомнить откровенную и циничную дискриминацию по возрасту при приёме на работу во многих российских фирмах.

Ещё несколько часов ушло на настройку альтернативного компилятора GNU C++ и — о, чудо! — получение нормального результата без всяких утечек для того же теста. К сожалению, прежний компилятор от IBM этим происшествием нас разочаровал полностью, это была капля, переполнившая чашу, ставшая причиной исключения виновника из арсенала. Но радоваться было рано. Другая используемая библиотека просто не имела официальной версии под AIX для нашего альтернативного компилятора, в наличии была только версия, сделанная «репрессированным» нами IBM-овским транслятором. Исходные тексты библиотеки также не содержали файлов инструкций (*makefiles*) компиляции, поэтому пришлось создавать их шаманскими способами, то есть собственными руками, подбирая опции по «образу и подобию». Наконец, успешно скомпилировав модули, мы не смогли их запустить, так как одна парочка, не выдавая никакой содержательной диагностики после запуска, тут же завершала свою работу по ошибке.

В итоге мы задержались с этим процессом до половины девятого вечера, поэтому, придя на станцию, обнаружили неприятный сюрприз: станция фактически закрылась, а поезда кончились ещё час назад. Собственно говоря, станция представляла собой домишко и две длинные платформы. Телемониторы, на которых обычно показывают четыре-пять ближайших поездов, были выключены, а домик закрыт на замок, и металлические жалюзи опущены. Поскольку мы не сразу обнаружили отсутствие поездов, то сначала купили билеты. Автомат по продаже был один, и он сильно напоминал старый аппарат по продаже топлива для «пепелацев» из культового кинофильма «Кин-дза-дза». Набрав номер станции назначения, я услышал скрип и с содроганием сунул в кривую прорезь кредитную карточку. После минутной паузы автомат издал визгливый звук — это

модем пытался соединиться и отправить сведения об оплате в свой центр. Ещё секунд через 30 из самого нижнего окошечка выпал жёлтый билет, который я засунул в другую неровную прорезь для компостирования. Наконец, выждав ещё секунд 20, зловеще поскрежетав иголками принтера, автомат выплюнул отмеченный билетик все из того же нижнего окошечка. Я слегка присел перед ним, расставив руки, и сказал «Ку!».

То, что народу на станции было полтора человека, нас не насторожило, это обычное дело. Но вот погасшие мониторы были плохим признаком. Наконец, после поисков мы увидели на стенке домика болтающийся листок, оказавшийся ксерокопией расписания и поведавший нам горькую правду. Не теряя времени, мы поднялись к шоссе в надежде сесть на автобус и доехать до узловой станции, где поезда ещё ходили. Но и там нас ждало разочарование: последний из них ушёл ещё в половине восьмого вечера. Смеркалось, вокруг практически ни души. Дима решил использовать преимущества мобильной связи и вызвать такси. Однако диспетчер после долгого выяснения, где же мы все-таки находимся, сказала, что этот район не обслуживается, после чего связь прервалась. Засунув телефон обратно в карман, Дима предложил идти до узловой станции пешком: хотя карты у нас не было, но он вроде как помнит примерное направление. По нашим расчётам станция находилась в трех-четырёх километрах, поэтому мы двинули в путь.

Через сорок минут быстрой ходьбы по практически безлюдным улицам мы вышли к цели и, купив по «Сникерсу» в другом автомате, сели на отходящий в 22:08 полупустой поезд на Париж. Я попытался воспользоваться Диминым телефоном и позвонить домой, дабы предупредить жену о вынужденной задержке вследствие наших мытарств, но голос оператора сообщил, что денежки практически закончились и сделать вызов нельзя. Дима решил сразу же пополнить счёт по карте, однако и здесь нас ждало разочарование: сервер по обслуживанию платежей был «на обслуживании», или, попросту говоря, не работал. До дому я добрался к полуночи, но мы договорились, что приедем завтра на час позже, чтобы немного отдохнуть.

Назавтра, в поезде, Дима опять пытался зарядить мобильный телефон порцией денег с «Визы», но сервер все ещё не воскрес, и телефон мог только принимать периодические звонки Блеза, который внезапно обнаружил серьёзную проблему в одном из модулей сопряжения с администраторским интерфейсом. По приезде в офис нас ждал новый сюрприз: вчерашний день был по плану последним, тогда как начальство забыло предупредить IBM-овцев о том, что мы ещё поработаем немного. Поэтому мы застали все временные каталоги на

нашей машине чистыми от файлов после регулярной утренней процедуры «уборки мусора за клиентами». Но это не было большой бедой, через час Блез примчался на своем мотоцикле с кассетой, и мы благополучно восстановили файлы недельной давности, а ещё за пару часов повторили с ними все вчерашние манипуляции. Но модули в новой редакции все так же не работали, как и вчера. Ну что же, опять чтение документации в Интернете, шаманство с опциями компиляции, сравнение работающего примера с неработающим модулем... Наши неутомимые поиски прервал менеджер, который попросил нас в целях соблюдения режима безопасности (интересно, а вчера что, день особый был?) закончить работу в шесть-тридцать вечера. Мы двинулись домой, что называется, несолоно хлебавши, но зато нас провели коротким путём через помещения завода. Я обратил внимание на многочисленные большие залы с IBM-овскими вычислительными машинами, представлявшими собой металлические шкафы размером с платяной. Рядом высились стойки с магнитными лентами. Я даже ощутил подобие ностальгии по безвременно разрушенным и разворованным на цветные металлы большим машинам у нас в СССР. А тут они ещё работали, и никто пока их выбрасывать не спешил.

Третий день начался ударными темпами. Удалось, буквально ткнув пальцем в небо, выяснить причину ошибки в модуле. Ею оказался настолько искусно криво написанный цикл в одной из многочисленных функций, что не сразу была видна его нехорошая тенденция при определённых условиях превращаться в бесконечный. Зная привычку авторов копировать текст, мы нашли и исправили пару таких же мест и в других функциях. Тем не менее модули все ещё не запускались в штатном режиме, но за пару часов экспериментов мы выяснили, что если некоторые библиотеки подключить статически, а не динамически, то всё в итоге будет жить нормально. Время как раз подходило к вечеру, мы оперативно отослали Блезу, который сидел у заказчика и, заговаривая ему зубы, ждал результатов, исправленный модуль и приступили к записи на диск наработанных за последние дни файлов. Провозились с этим до семи часов вечера, а дело было в пятницу, когда народ старается уйти с работы пораньше. В зал опять зашёл менеджер, осведомился о планах на вечер и, услышав, что всё сделано, обрадовался и даже предложил подвезти нас до города.

Хорошо там, где нас нет

Читатель может справедливо подумать, что пример нашей конторы не показателен. Она маленькая, да и специалистов сейчас повсеместно не хватает, они

дороги, а для небольших фирм зачастую недоступны. Однако с подобными проблемами сталкиваются практически все мои знакомые и коллеги. Если фирма с 50 % национального рынка электронных платежей вполне может работать без системы управления версиями исходного кода и дублированием таблиц в базе данных, то что же тогда говорить о стартапе...

Зато вы можете быть спокойными: без работы в ближайшие десятилетия не останетесь. Некоторые мои друзья в США жаловались на некоторую неопределённость в связи с разыгравшимся там кризисом лопнувших «дот-комов». Но вряд ли следует бояться заразиться насморком во время эпидемии чумы.

О технических книгах

Дефрагментация мозгов

Современное софтостроение заслуженно забыто наукой. Ну а что вы хотите, если на вопрос «Почему нет науки на конференциях и в публикациях?» получаешь однозначный ответ «Никакая наука рынку не нужна»¹. В результате на первый план выходят повара, написавшие очередную порцию рецептов приготовления пиццы и винегрета из найденных в холодильнике продуктов корпораций.

Ошибочно полагать, что такая ситуация возникла недавно. Отрицательная динамика степени полезности технических книг наблюдалась уже в начале 1990-х годов. Например, С. Дмитренко в предисловии к своему переводу известной книги Leo Brodie «Thinking FORTH. A Language and Philosophy for Solving Problems» в 1993 году писал:

Можно сказать много грустных слов о тенденциозности современной околотехнической литературы, о переориентации отечественных программистов и разработчиков с исследовательских и новаторских на чисто коммерческие работы, о складывающемся монополизме отнюдь не лучших (зато более хватких) производителей компьютеров и программного обеспечения и т. д. И все же, несмотря на эти признаки нарастающего вырождения в нашем, да и мировом компьютерном деле, я надеюсь, что подъем наступит, и мы будем его свидетелями и реализаторами.

¹ См., например, заметку обозревателя PC Week/RE А. Колесова «Научные конференции возвращаются в ИТ».

Нынешняя «гуглизация» позволяет быстро находить недостающие фрагментарные знания, зачастую забываемые уже на следующий день, если не час. Поэтому ценность книг как систематизированного источника информации, казалось бы, должна только возрастать. Действительно, автору бессмысленно брать на себя функции интеллектуального фильтра RSS¹, составляя компиляции из содержимого чужих блогов. Кроме стоящих вне конкуренции учебников остаётся в основном только конкретизация — узкоспециализированная проработанная технологическая тема либо обобщение концептуальных наработок и практического опыта.

Но для возрастания значения книг на фоне всеобщей фрагментации знаний необходимо умение аудитории читать и усваивать длинные тексты, неуклонно снижающееся последние десятилетия.

Как-то раз сидели мы с хорошим знакомым на террасе его уютного дома и, потягивая из железных трубочек чай-мате, обсуждали, ни много, ни мало, но смену парадигмы мышления. Долгий опыт университетского преподавателя Павла Андреевича Калугина ныне обогатился добавлением в курс информатики такой эклектики, как Enterprise Java. Проблемы подобные курсы вызывают больше не у студентов, а у обладающего системным образованием и мышлением преподавателя, вынужденного каким-то образом связывать противоречивое и выстраивать логику там, где её изначально не было.

Почему же относительно легко студентам? Дело, прежде всего, в смене образа мыслей, а может быть, и восприятия самой действительности. Мир — это такая очень большая и сложная компьютерная игра, а преподаватели, соответственно, должны обучать не премудростям стратегий познания мира, а практическим приёмам, секретным кодам и даже шулерству, чтобы пройти в этой игре на следующий уровень. Этакие «мастера ключей» из Матрицы. Про то, что в игре продукты на столе появляются прямо из холодильника, тоже стоит упомянуть.

Изредка мне приходится практиковать обучение СУБД, где я столкнулся ровно с тем же явлением. Группа стажеров примерно моего возраста и старше всю неделю честно старалась вникнуть в детали, написанные мелким шрифтом после каждого слайда, уместить знания в некую систему, желательно, не диссонирующую с уже имеющейся в голове.

И напротив, аудитория примерно 20–25 лет оказалась совершенно равнодушной к пояснительному тексту. Если картинка слайда была удачной, то она более-

¹ Агрегация в единый поток информации из множества источников: анонсов статей, изменений в блогах и т. п.

менее откладывалась в памяти. Вдобавок, шло конспектирование некоторых случаев из моей практики с короткими кусками кода. Иное дело — лабораторные работы, когда решение находилось увлекательным методом «научного тыка». Если же метод не срабатывал, то ко мне обращались с просьбой подсказать «код доступа для прохождения на следующий уровень».

Много копий сломано в обывательских дискуссиях о так называемом клиповом мышлении, шаблонности, «плохом образовании» вкупе с апокалиптическими прогнозами и прочим. Хотя на самом деле пока непонятно, к чему приведёт тенденция в перспективе ближайших десятилетий. Во фрагментарном «клиповом» мышлении есть и свои плюсы: способность быстро решать типовые задачи, широкая квалификация и мобильность, меньшие затраты на массовое образование. Минусы тоже ясны. В апокалипсис технократии я не верю, всегда будут рождаться способные дети и существовать ограниченное число учебных заведений, дающих жутко дорогое фундаментальное образование, вроде того, что было общедоступным в СССР. Видимо, этого должно хватить на поддержку критичных технологий цивилизации и дальнейшее их развитие.

Является ли фрагментарное мышление приспособлением к многократно увеличившемуся потоку информации? Отчасти да, только это скорее не адаптация, а инстинктивная защита. Чтобы осознанно фильтровать информацию о некоторой системе с минимальными рисками пропустить важные сведения, нужно иметь чётко сформированные представления о ней, её структуре и принципах функционирования. Если, например, у стажера нет знаний о СУБД в целом, то курс по SQL Server — конкретной её реализации, сводится к запоминанию типовых ситуаций и решений из набора сопутствующих практических работ. Вся теоретическая часть при этом просто не проходит фильтры.

В такой ситуации альтернативой ухода от информационных потоков и некачественного образования во фрагментарную реальность является самообразование на базе полезных книг. Потому что хорошая книга — самый эффективный способ дефрагментации ваших мозгов. Если не верите, возможно, вас убедит рассказ классика научной фантастики А. Азимова «Профессия» о неудавшемся программисте.

Как вы заметили, я упомянул о полезности книг, но никакого объективного определения этому критерию не дал, хотя сам же и сетовал на недостаток науки. Не претендуя на приоритет и оригинальность, поделюсь следующими несложными правилами и эвристиками.

Простые правила чтения специальной литературы

Хороших и полезных книг в принципе не может быть много. Иначе вы посвятите все рабочее время чтению. Тут уместно вспомнить второе правило Аникеева [12]; кстати, все три его правила стоит привести целиком. Итак, настоящий исследователь должен:

- быть достаточно ленивым. Чтобы не делать лишнего, не ковыряться в мелочах;
- поменьше читать. Те, кто много читает, отвыкают самостоятельно мыслить;
- быть непоследовательным, чтобы, не упуская цели, интересоваться и замечать побочные эффекты.

Может показаться странным, что в теме о технической книге я сослался на художественное произведение. Будучи студентом, я однажды услышал от заведующего нашей кафедры вычислительных и радиоэлектронных систем профессора и академика Михаила Борисовича Игнатьева совет, касавшийся обычной лабораторной работы, связанной с телеконференциями. Дословно он звучал так: «Я бы посоветовал Вам почитать Станиславского. Телеконференции — это большое представление, шоу...». Если рекомендацию тогда я всерьёз не воспринял, то сегодня оцениваю её как весьма полезную.

Когда натыкаешься в Сети на очередное несдержанное восклицание «*must have*», подобная постановка вопроса коробит сама по себе. Представьте, что кто-то открыл для себя и полюбил варёный лук. И теперь если ты его не «*must have*», то как бы и не в струе, и вообще от жизни отстал. Смешно, конечно, но шутки шутками...

Технические книжки — это не бессмертные произведения графа Толстого. К ним должен быть суровый, сугубо индивидуальный и безжалостный подход без какой-либо оглядки на чужое мнение. Прежде чем начать читать, возьмите листок бумаги, оптимально формата А4 — сложенный пополам, он удачно вкладывается в книгу, служа одновременно и закладкой. И запаситесь карандашом. Пользователям устройств для чтения электронных книг, к коим я тоже отношусь, необходимо освоить функции вставки в текст своих заметок.

Если «читалка» этого не позволяет, придётся всё-таки обратиться к бумаге или сменить устройство на более функциональное.

Теперь, по мере чтения, начинайте делать пометки в стиле «с. 11, это мнение плохо обосновано, потому что...», «стр. 25, хорошая мысль, применимо также в ситуации...». Очевидно, для пометок типа «у автора N выводы другие, но если обобщить...» потребуется немного больше времени.

Если, пролистав с полсотни страниц, вы не сделаете ни одной пометки, то можете смело закрывать сие произведение. В топку бросать, наверное, не надо, но поберегите своё время. Если же к концу прочтения вам хватит листа А4, исписанного убористым почерком, для всех пометок, то можете считать, что чтение прошло не зря. Ну а если одного листа не хватило... Тогда смело идите на свой любимый интернет-форум и там поделитесь с коллегами своими находками. Когда вместо малоинформативного «must have», используя все тот же испещрённый пометками лист, вы сможете передать пользу от прочитанного, коллеги будут вам чрезвычайно признательны за рекомендации и собственное сэкономленное время.

Литература и программное обеспечение

Как вы знаете, в софтверостроении, часто употребляется термин «(на)писать программу». Первые программы для ЭВМ не писали, а составляли. Составляли из машинных команд. Кодировали. Пробивали дырочки на картонных перфокартах. Затем появились языки высокого уровня, приближенные к естественным. Тогда программы стали именно писать. Сейчас, когда отрасль предпринимает попытки индустриализации, можно услышать, что программы вновь кодируют. И снова это связано с разделением труда: кодируют по спецификациям проектировщиков или по наитию пользовательских историй.

В литературе выражение «закодировать» пока не прижилось, но налёт индустриальности уже коснулся и этой области человеческой деятельности. Миллионы журналистов ежедневно вставляют свои модули в разные форматы периодических изданий: от экспертных авторских колонок до банальной «джинсы»¹. Здравницы и некрологи пишут загодя. Стереотипная массовая литература вроде карманных детективов и любовных романов поставлена на поток.

¹ Джинса (ударение на последнем слоге) — жаргонное название заказной статьи.

Разделение труда на «автора» и «кодировщика» в подобном процессе также присутствует. Даже если автор номинальный, издающий под своим брендом труд неизвестных литераторов, что, к сожалению, случается.

В итоге обнаруживаем немало сходства в процессе. Взглянем на классификацию создаваемых продуктов, плодов труда, приведённую в таблице.

Литература		Софтостроение	
Формат	Содержание	Формат	Содержание
Рассказ	Эпизод из жизни общества (сообщества)	Программа	Решение частной задачи
Повесть	Связные эпизоды из жизни общества (сообщества)	Программный пакет	Решение нескольких задач, как правило, связанных
Роман	Жизнь общества (сообщества)	Программная система	Решение задач предметной области
Роман-эпопея	Жизнь народа	Программный комплекс	Решение задач нескольких связанных предметных областей

Сходство усиливается. Обратите внимание на технологию использования продуктов. Исполнительной средой для программы является ЭВМ. Для литературного произведения в таковой роли выступает наш мозг. Глубокое погружение человека в процесс чтения соответствует 100 % загрузке процессора, а то и даже «зависанию». Откладывание книжки после пары просмотренных страниц говорит о несовместимости программы с текущей конфигурацией исполнительной среды.

Настало время посмотреть на архитектуру продуктов. Прежде всего, на степень ее абстрактности.

Существуют программные системы, представляющие собой весьма общее решение, по сути, каркас, фреймворк, который другой программист может использовать для решения своих задач. В литературе подобные фреймворки тоже встречаются. Как правило, на них ссылаются: «Это сюжет, основу которого составляет...». Наиболее известным разработчиком фреймворков является Вильям Шекспир. Базовая подсистема уровня ядра «Быть или не быть», обёрнутая собственным кодом писателя, присутствует практически в любом про-

изведении, претендующем на глубину. Если придерживаться аналогий, Линус Торвалдс — Шекспир современного мира операционных систем на базе Linux.

В противоположность каркасам, встречаются самодостаточные программы, у которых, как говорится, не убавить и не прибавить. Законченные продукты решают поставленные пользователем задачи. Их надо заключать в рамочку хрестоматийных произведений, используя в соответствии с собственной программой обучения, например школьной.

Вспомним и о таком важном свойстве, как открытость архитектуры.

Программы с открытой архитектурой позволяют создавать свои встраиваемые модули расширений¹ и легко интегрируются с другими программами. Не нарушая общей логики, совсем нетрудно взять пару ключевых героев и дописать эпизод из их жизни, не вошедший в книгу.

Монолитные же программы закрыты. Стоит потянуть за одну ниточку, как сдвинутся целые пласты. Только автору под силу внести дополнения в свое произведение. Допisać самостоятельный «приквел» или «сиквел» к «Войне и миру» — задача практически невыполнимая.

Наконец, существует как пакетная работа с программами, так и диалоговая.

Пакетная обработка позволяет результат одной программы использовать для другой. Неважно, что главного героя увозят на тарелке инопланетяне. Заводим нового, помещаем его в старые декорации — вот и готово продолжение.

В диалоговом режиме все ненужные подробности скрывает красивый интерфейс, как правило, графический. Это не только комиксы и детские книги с обилием иллюстраций. Вполне можно обойтись текстовым режимом, лаконично излагая суть и делая сноски и ссылки для любознательных.

Хотя софтверное строительство довольно часто сравнивают со строительством домов, видимо, желая поскорее свести роль программистов к укладке готовых кирпичей и тем самым закрыть надоевшую проблему огромного числа просроченных или неудачных проектов, не бойтесь аналогий с писательским трудом. Ведь разработка программ совсем не означает необходимость всякий раз создавать «нетленку». В конце концов, хороший *plug-in* во много раз полезнее плохого фреймворка.

¹ Англоязычные аналоги *plug-in*, *add-in*, *module*.

Вместо послесловия, или Краткое изложение «Оснований»

В удивительные и непредсказуемые времена
мы в испуге цепляемся за прошлое.

А. Азимов. «Край основания»

Вначале, как известно, было Слово™. И Слово™ содержало ровно столько байтов, сколько допускала архитектура ЭВМ. Поэтому 1 К не был равным 1 кбайт.

Из Слов™ стали составлять настолько скучные и однообразные на вид инструкции, что выполнять их могла только ЭВМ. Или самые упорные из составителей — в своем уме. И называли эту технологию Программированием© в Машинных Кодах™. Вскоре парни, вооружённые Программированием в Машинных Кодах™ сделали Первую© Систему®. Далее значки торговых марок и копирайта я опускаю.

Когда парни сделали Первую Систему, их стали нагружать внесением в неё новых функций. Парни настолько были заняты этим делом, что у них едва хватало времени на чашку кофе и сигарету. Кроме того, они несколько состарились за работой и стали Старыми Чудаками.

И тут появились Молодые Парни, которым пришла в голову Мысль: можно вместо Машинных Кодов использовать их Мнемонику. Понятную человеку. Но, к сожалению, непонятную ЭВМ. Пришлось им в Машинных Кодах составлять вспомогательную программу — Транслятор, которая переводила Мнемокод в Машинный Код. На самом деле мысль Молодые Парни подслушали у Старых Чудаков, но тем, как мы помним, было не до реализации этой Мысли. И Молодые Парни, вооружённые Программированием в Мнемокодах, сделали Вторую Систему.

Когда Молодые Парни сделали Вторую Систему, их, как вы догадались, снова стали нагружать внесением в нее новых функций. Вначале парням удавалось вносить изменения так быстро, что оставалось время на несколько чашек кофе в день. И тогда их стали напрягать больше. Теперь Молодые Парни настолько были загружены, что у них опять едва хватало времени на чашку кофе и сигарету. Кроме того, они тоже состарились за работой и стали Старыми Чудаками.

Во время питья одной из чашек кофе у Старых Чудаков возникла мысль о том, что хорошо бы приблизить составление программы к написанию текста на человеческом языке. На Языке Высокого Уровня (ЯВУ). Но Чудаки были так заняты Второй Системой, что благородно позволили подслушать эту Мысль новым Молодым Парням, подсматривавшим за ними сквозь дырки в перфокартах.

Молодые Парни поднатужились и составили Транслятор с языка, похожего на математическую запись формул. Его так и называли: Фортран — ФОРмуло-ТРАНСлятор. Вооружённые Фортраном, Молодые Парни сделали Третью Систему. Но не успели они, быстренько добавив в Систему новые функции, сесть за кофе и партию в преферанс, покуривая в форточку подсобки, как их снова сильно напрягли. И снова стали они Старыми Чудаками. Да, чуть не забыл: в соседней комнате другие Молодые Парни и Девушка создали другой Транслятор, КОБОЛ — Как бы Описания Бизнес-правил... Ой, ладно. Они тоже делали Третью Систему, но с другого боку.

Надо сказать, что Старые Чудаки с Фортраном придумали такую вещь, как типы переменных по умолчанию. И все переменные, начинающиеся на i , j , k , l , m и n Транслятор считал целочисленными. С той поры все Новые Молодые Парни в своих Новых Системах используют эти переменные в циклах, особо не задумываясь об их происхождении.

Третья Система оказалась такой живучей, что до сих пор можно увидеть объявления о поиске ну-Очень-Старых-Чудаков для внесения в нее изменений.

Очередные Молодые Парни стали кривить лица и указывать на сложность ЯВУ и несовершенство Трансляторов. Покривившись, они приступили к созданию более простых, ясных и надёжных ЯВУ, как им тогда казалось. Четвертую Систему парни начинали делать на С, а прикладную часть к ней — на Паскале. А сделав, как водится, превратились в Старых Чудаков. Причём одни Старые Чудаки так и остались на С, а другие потом решили добавить к названию два плюса. Третьи, поэкспериментировав с приставками «Турбо», вообще изменили название в пользу Дельфийского Оракула.

Следует вспомнить, что Один Умный Старый Чудак из Швейцарии потом продолжил создавать Следующие Системы. Но его упорно не замечали. И не потому, что его Системы были плохи. А потому, что все были по горло заняты внесением новых функций в Четвертую Систему.

Народу, вносящего изменения, все прибывало. Началось Вавилонское столпотворение ЯВУ и Подходов к Проектированию Программ. Каждый Молодой Парень считал свой ЯВУ лучшим, а Подход — единственно правильным. Но, сделав свою собственную Четвертую Систему, каждый парень становился очередным Старым Чудаком и неизбежно погрязал в добавлении в нее новых функций.

Наконец, Большие Дяди сказали себе, что пора бы навести Как бы Порядок в этом Творческом Бардаке. И стали вкладывать много денег для создания Единственно Правильной Четвертой-с-половиной Системы. Она не должна была зависеть от типа ЭВМ: единожды написанная на ЯВУ программа могла бы работать на любой другой ЭВМ. А чтобы новые Молодые Парни не создавали хаос, Большие Дяди создали свой Стандарт на ЯВУ и всё, что вокруг. На практике же оказалось, что однажды написанная программа должна быть отлажена везде, где её планируют запускать. Ну а чтобы другим парням и чудакам было не обидно заниматься отладкой и без передышки вносить в Четвертую-с-половиной Систему изменения, назвали её просто «Кофе».

Когда Большие Дяди стали поднимать Большие Деньги на своей Четвертой-с-половиной Системе, то Другие Большие Дяди решили, что они не хуже. И тоже сделали Четвертую-с-половиной Систему. Но, поскольку название «Кофе» оказалось занято, а «Сигарета» явно не соответствовала маркетинговым концепциям, её назвали так, чтобы все были бы внутренне горды от одного только чувства того, что работают они с Великой Четвертой-с-половиной-и-ещё-Точка Системой.

Внесение изменений в Четвертые-с-половиной Системы стало занятием столь же скучным, как и предыдущие. Поэтому, чтобы Молодые Парни не чувствовали себя, по крайней мере сразу же, такими же Старыми Чудаками, им разрешили обзывать всех вносящих изменения в Четвертую (ту, что без половины, а тем паче в Третью) Систему Старыми-Чудаками-на-Одну-Букву.

И вот одни Старые Чудаки добавляют функции в Четвертую, Третью и даже Вторую Системы. Другие — в Четвертые-с-половиной Системы. А в перерывах на форумах и в блогах последние обзывают первых Одной Буквой.

Но, поскольку форумы и блоги открыты, за процессом наблюдают Самые Молодые Парни. И, видя это, им всё меньше хочется делать Пятую Систему. Во-

первых, надо «учиться, учиться и ещё раз учиться» и отнюдь не на трёхмесячных курсах, а во-вторых, застолбившие рынок Большие Дяди не собираются его ни с кем делить. Но и становиться Старыми Чудаками при Четвертой-с-половиной Системе, минуя стадию Молодых Парней, им тоже не особо хочется.

И тогда все хором — от Старых Чудаков Первой Системы до Самых Молодых Парней — сказали: в нашей отрасли кризис! А Старые Чудаки добавляют: застой, и света в конце тоннеля не видно. Рядом с ними Маргинальные Чудаки с искусственным интеллектом под мышкой, специализированными ЯВУ и ЭВМ, суперкомпьютерами, векторными вычислениями и прочей экзотикой стоят в сторонке от мейнстрима и посмеиваются над ловкостью, с которой им удалось обмануть всех, включая самих себя.

Литература

1. *Брукс Ф.*, Мифический человеко-месяц или как создаются программные системы. СПб.: Символ-Плюс, 2001. 304 с.
2. *Фокс Дж.*, Программное обеспечение и его разработка. М.: Мир, 1985. 368 с.
3. *Бир Ст.*, Кибернетика и менеджмент. М.: КомКнига, 2006. 280 с.
4. *Nicolas Carr*, The Big Switch: Rewiring the World from Edison to Google. W. W. Norton & Company, 2008.
5. *Васкевич Д.*, Стратегии клиент/сервер. Киев: Диалектика, 1996.
6. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.*, Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2007. 366 с.
7. *Гладких Б. А.* По волне моей памяти // Вестник Томского гос. ун-та, 2002. № 275. С. 249–258.
8. *Тарасов С.*, Обзор средств объектно-реляционной проекции (ORM) для платформы .NET, ZDNet.ru, октябрь 2005. Текст доступен по ссылке <http://arbinada.com/main/node/33>
9. *Тарасов С.*, Разработка ядра информационной системы // Мир ПК. 2007. № 7.

10. Серия статей «Информационная система Ниеншанц» // Компьютер-Информ. 1997. № 13–15.
11. *Донской М.*, Жизненный цикл программиста, статья-эссе для сайта polit.ru, 2008 г.
12. *Гранин Д.*, Иду на грозу. М.: Молодая гвардия, 1966.
13. *Кольвах О., Копытин В.*, Адаптивные модели бухгалтерского учета и формирования финансовой отчетности в системе кредитных организаций. М.: Терра, 2000.
14. *Хеннинг М.*, Восход и закат CORBA, ACM Queue, Volume 4, Number 5, June 2006, перевод С. Кузнецова, citforum.ru
15. «Joint Strike Fighter Air Vehicle. C++ coding standards for the system development and demonstration program», Lockheed Martin Corporation, December 2006.
16. *Jason Weiss*, Is complexity hurting Java? // Java Developer's Journal. Vol. 7, Issue 10. Octobre. 2002.
17. *Уэзерелл Ч.*, Этюды для программистов. М.: Мир, 1982. 288 с.
18. *Кривошеин М.*, Введение в складской учет. 2002, 2005 (испр. и доп.). Текст доступен по ссылке: <http://arbinada.com/main/node/15>
19. *Иванов В., Тарасов С.*, Как проектировать бухгалтерию?, 1999, 2005 (испр. и доп.). Текст доступен по ссылке: <http://arbinada.com/main/node/16>
20. *Тарасов С.*, Уровни изоляции транзакций в SQL // Мир ПК. 2009. № 7.
21. *Кулигин В. А., Кулигина Г. А., Корнева М. В.*, Кризис релятивистских теорий. Доклад на международном конгрессе «Фундаментальные проблемы естествознания и техники», С.-Петербург, июль 2000 г.

22. *Новиков Л.*, Введение в Rational Unified Process. М.: Интерфейс, 2000.
23. *Capers Jones*, «Estimating Software Costs: Bringing Realism to Estimating», Second edition, McGraw Hill, April 2007.
24. MDD. Разработка, управляемая моделями, IBM Developer Works, IBM, 2007.

Сергей Тарасов

Дефрагментация мозга. Софтостроение изнутри

Заведующий редакцией

А. Кривцов

Руководитель проекта

А. Кривцов

Ведущий редактор

Ю. Сергиенко

Литературный редактор

Е. Пасечник

Художественный редактор

Л. Адуевская

Корректор

И. Тимофеева

Верстка

Л. Родионова

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 22.03.13. Формат 70х100/16. Усл. п. л. 23,220. Тираж 1200. Заказ
Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?

Вы ведете блог?

Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?

ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!



Зарегистрируйтесь на нашем сайте в качестве партнера по адресу www.piter.com/ePartners



Получите свой персональный уникальный номер партнера



Выбирайте книги на сайте www.piter.com, размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт www.piter.com)

ВНИМАНИЕ! В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:

<http://www.piter.com/book.phtml?978538800282> – обычная ссылка

<http://www.piter.com/book.phtml?978538800282&refer=0000> – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

**Подробнее о Партнерской программе
ИД «Питер» читайте на сайте
WWW.PITER.COM**

**ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM**



ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным
направлениям: история и публицистика, экономика и финансы, менеджмент
и маркетинг, компьютерные технологии, медицина и психология.

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электrozаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: nnovgorod@piter.com

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: samara@piter.com


УКРАИНА


Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com


Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: minsk@piter.com

 Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых
партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: spb@piter.com

 Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

 Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: uchebnik@piter.com

 Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225





КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **postbook@piter.com**
- по телефону: **(812) 703-73-74**
- по почте: **197198, Санкт-Петербург, а/я 127, ООО «Питер Мейл»**
- по ICQ: **413763617**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

-  Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа Вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате все виды электронных денег: от традиционных Яндекс.Деньги и Web-money до USD E-Gold, MoneyMail, INOCard, RBK Money (RuPay), USD Bets, Mobile Wallet и др.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения Вами заказа.

Все посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку Ваших покупок максимально быстро. Дату отправления Вашей покупки и предполагаемую дату доставки Вам сообщат по e-mail.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.



Нет времени ходить по магазинам?



наберите:



www.piter.com



Здесь вы найдете:

Все книги издательства сразу
Новые книги — в момент выхода из типографии
Информацию о книге — отзывы, рецензии, отрывки
Старые книги — в библиотеке и на CD



**И наконец, вы нигде не купите
наши книги дешевле!**