Навыки элегантного программирования. Часть 2.

Михаил Зускин | 06.10.2010

Продолжение. Начало статьи здесь.

2. Именование переменных

«И назвал Бог свет днем, а тьму ночью.»
Первая книга Моисея

Точный смысл

Давайте своим переменным (а также классам и методам, но начните с полей таблиц баз данных!) осмысленные названия, и *тогда в большинстве случае в комментариях не будет необходимости*. Широко применяйте слова «per» и «by» – они действительно облегчают жизнь разработчика! Название переменной «cowsPerFarm» лучше чем «totalCows», а название метода «RetrieveCityByCountry» скажет нам больше чем «retrieveCity», особенно если у него нет параметров, предоставляющих информацию о критериях поиска. Кроме того, такие слова значительно упрощают написание и чтение финансовых расчетов! Не применяйте абстрактные слова типа «actual» и «total» в полях баз данных и именах переменных. Они доведут вас до бешенства и заставят потратить дополнительные часы (за время разработки проекта) в попытках понять, что означает «actual» и «not actual» и какому уровню группировки соответствует «total». Дорогой читатель, скажи мне прямо сейчас: поле «totalCows» содержит сумму на уровне коровника, фермы, деревни, области, страны, Вселенной? К примеру, если речь идет о ферме, то как вы назовете «total» для деревни? *Это тоже «total», не так ли?* Такие слова бессмысленны без контекста. Он очевиден, когда архитектор данных создает таблицу («это же просто – общее количество на территорию, о которой я думаю прямо сейчас!»), но впоследствии неясен – порой очень трудно понять контекст, взирая на переменную, окруженную сотнями строк кода. Поэтому не пишите «total», пишите «total PER WHAT»! Только точные определения, (почти) не вызывающие вопросов, даже если это выливается в более длинные названия полей и переменных! Если не вы создаете объекты баз данных, распечатайте этот параграф и пришпильте его на рабочем месте вашего администратора или архитектора баз данных.

*** ПЛОХОЙ код: ***

```
totalSalary = actualSalary * totalHours; // :-(
```

*** ХОРОШИЙ код: ***

```
salaryPerDay = salaryPerHour * hoursPerDay;
```

Постоянство

Используя значения, извлекаемые из таблиц баз данных, старайтесь давать соответствующим переменным те же имена, что у полей таблиц (конечно, добавляя префиксы по соглашению об именах). Это позволит избежать различных имен для одной и той же сущности. Никаких ‘empId’ и ‘employeeId’ в одном приложении! Только так, как это выглядит в таблице. Но предыдущее правило более важно, особенно если вы пишите метод с финансовыми расчетами, а поля таблиц именованы, словно их создатель не знал других слов кроме «actual» и «total»…

Никаких аббревиатур

Пытаясь уменьшить размер наших скриптов, мы в большинстве случае сокращаем слова при именовании таблиц, колонок, объектов приложения, скриптов и переменных. Иногда мы укорачиваем их незначительно, иногда – существенно. Обычно это зависит от практики, существующей в компании, на которую вы работаете. На одном из моих рабочих мест я страдал от очень жестких правил сокращения. Они сокращали все, что видели. Не использовали

гласных, за исключением ведущих (wrk – work, crtfc – certificate, insr – insurance и т.п.). Были и другие официальные правила отрезания мяса из слов и оставления только костей скелета. Когда я не мог понять одно из таких «слв» (а это случалось часто), я открывал специальный список аббревиатур! Такие имена были понятны только древним Египтянам, но не современным разработчикам, пришедшим в проект. Когда мой контракт завершился, я был счастлив! Конечно, мы стремимся сокращать, но зачем так усложнять жизнь? Когда я начал работать по одному из моих недавних контрактов, я увидел обратную картину: никаких сокращений! ОК, иногда (хорошо, простые случаи вида «id», «sys», «col» или «num»), но обычно полные предложения, применяемые в качестве имен таблиц, полей, методов и т.п. Я читал скрипты как книгу приключений, а не как код на языке программирования! Первоначально, я был слегка шокирован: как они это делают? Неужели они не знают, что каждый разработчик должен делать свои скрипты как можно короче? Это не соответствовало тому, что я видел в других проектах за много лет моей карьеры… Но я хочу вам сказать: в этом проекте было действительно приятно работать! Сочтите это шокирующим и необычным, но вот отличная идея: НИКОГДА не сокращайте слова в названиях ваших объектов! Конечно, этому совету могут следовать только ребята, создающие новые системы (включая объекты баз данных)… Если слова в системе, над которой вы работаете, уже покалечены, ничего не попишешь – применяйте имена в их существующем виде и не создавайте других версий для тех же самых сущностей (постоянство превыше всего!).

Длинные, наглядные имена

Не бойтесь давать своим переменным и методам длинные имена на реальном английском языке, которые описывают их назначение (я называю это «стилем Льва Толстого»). Почувствуйте разницу между

```
farmingYears = this.GetFarmingYears(); // плохо...
```

и

```
farmingYearsWithAllowableExpenses = this.GetFarmingYearsWithAllowableExpenses(); // хорошо!
```

… если впоследствии переменная используется в сложной бизнес-логике. На самом деле переменная farmingYears лжет – у фермера могут быть годы без разрешенных затрат. В будущем это может привести к багам! Однажды мой коллега посмеялся над названиями моих переменных, но лучше смеяться над кодом, чем плакать (или даже оплакивать его), не так ли? Иногда названия моих переменных (большей частью, это булев тип) рассказывают целую историю, облегчая людям чтение моих скриптов. Разве это плохо? Даже если из-за этого строка будет слишком длинной, не составит труда разбить ее на две строки.

Точное определение времени действия

Давая имена переменным и полям таблиц, старайтесь, чтобы у будущего читателя не возникало вопросов, СЛЕДУЕТ ЛИ ВЫПОЛНИТЬ действие, или же действие УЖЕ ВЫПОЛНЕНО. Например, в одном из моих проектов у двух различных таблиц были поля ‘calculation_method’, но одно из них хранило значение, которое следовало применять в следующем расчете, а второе – уже использованное значение. Почему бы не назвать колонки соответственно ‘calculation_method_to_use’ и ‘calculation_method_used’, если это улучшит понимание бизнес-логики и SQL-запроса? А что вы думаете о переменной ‘isCalculation’? Расчет – чего? И что означает ‘if (isCalculation)’? Проблем с пониманием не будет, если переменная названа isCalculationDone (isCalculated) или в повелительном наклонении – doCalculation или calculate. Таким образом, простой совет: применяйте слова вида «do…», «…ToDo», «perform_…», «execute_…», «…Apply» и т.п. для того, которые будут происходить в будущем, и «…Done», «…Performed», «…Executed», «…Occurred», «…Passed», «…Applied» для того, что уже произошло.

Поднять флаг!

Не давайте булевским переменным «супер-универсальные» имена вроде ‘flag’ или ‘switch’. Лучше опишите В ЧЁМ ИМЕННО их назначение. Например, если аргумент метода разрешает или запрещает редактирование поля, назовите его ‘enableField’ (или ‘disableField’, но позитивная формулировка предпочтительней – она облегчает чтение вложенных логических выражений) – вы должны описать что аргумент делает или какую информацию передаёт, вы не должны объяснять другим программистам что такое тип boolean – поверьте, они знают!

Соглашение об именах

Применяйте соглашения об именах! Имхо, очень важно легко различать в коде:

- Локальные переменные (в т.ч. аргументы методов) и переменные экземпляров. Вам повезло, если ваша среда разработки (IDE) умеет раскрашивать переменные разной области видимости разными цветами или выделять их полужирным шрифтом или курсивом (как например, мой любимый Eclipse), но если не умеет, то вы можете, к примеру, следовать следующим правилам: localVariable, _privateInstanseVariable, PublicProperty (можно также добавить __privateStaticVariable).
- Простые переменные удобнее применять, если они снабжены префиксом в виде однобуквенного идентификатора типа данных: integer iAgeInYears, floating-point fPrice, boolean bMiracleOccurred, string sLastName, date dHireDate. Это правило также хорошо подходит для строк и дат в тех языках

программирования, в которых они представляют собой объекты, а не простые типы данных (к примеру, в C# и Objective-C).
- GUI-элементы различных типов. К примеру, lblLabel, txtTextBox, btnButton, dgDataGrid и т.п.
- Объекты различных классов в фреймворках вроде ADO.NET. Например, cnConnection, cmdCommand, dsDataSet, tblDataTable, rowDataRow, colDataColumn, drDataReader, daDataAdapter…

3. Мир баз данных

Только суррогатные первичные ключи!

При создании таблиц баз данных никогда не используйте в качестве первичных ключей поля, имеющие реальные значения в жизни! Даже если вы уверены, что значения уникальны и обязательны в бизнесе! Такие ключи называются «естественными ключами» («natural keys»). ВСЕ ПЕРВИЧНЫЕ КЛЮЧИ ДОЛЖНЫ БЫТЬ СЧЕТЧИКАМИ (1, 2, 3 и т.д.) И НЕ ИМЕТЬ ДРУГОГО КОНТЕКСТА КРОМЕ ИДЕНТИФИКАЦИИ ЗАПИСЕЙ! Такие ключи называются «суррогатными (синтетическими) ключами» («surrogate, synthetic keys»). Например, вы создаете информационную систему для компании, в которой у каждого сотрудника уже есть идентификационный номер. Не применяйте этот номер в качестве первичного ключа для таблицы ‘employees’! Вместо этого создайте другой, скажем, emp_id. Кроме этого, можно создать поле для хранения существующих идентификаторов сотрудников и привязать к ним ограничения NOT NULL и UNIQUE (то же самое для других «реальных» полей, скажем, Social Insurance Number). Почему бы и нет? Теория баз данных называет такие поля «ключами-кандидатами»: они подобны кандидатам на выбор в качестве первичных ключей, но не позволяйте им выиграть выборы! Многие архитекторы баз данных не осознают трудностей, возникающих у разработчиков при использовании естественных ключей! *Маленькая проблема*: когда в качестве первичного ключа в таблице присутствует комбинация из множественных полей (я видел 5-8!!!), разработчики вынуждены:

- Для идентификации записи объявлять и передавать между объектами несколько переменных и даже целые структуры/классы, хотя ОДНОЙ ПРОСТОЙ ПЕРЕМЕННОЙ вполне достаточно для выполнения этой работы.
- Создавать SELECTы с раздутыми кошмарными WHERE-конструкциями с нагромождением лишних строк, написанных только ради соединения двух таблиц. Важные бизнес-условия могут просто затеряться в такой неразберихе!

Но также есть *большая проблема*, возникающая при необходимости изменить информационную систему из-за изменений в бизнес-требованиях. Представьте: однажды заказчик прекращает заниматься идентификационными именами сотрудников. Другой пример: дубликаты номеров ID-карт, обнаруженные по крайней мере в одной стране, насколько я знаю. Вы будете изменять функциональность (отношения между таблицами, а также объекты баз данных и GUI-приложения), затрачивая время на повторную сборку и тестирование системы. Но если вы применили суррогатные первичные ключи, вам не потребуется выполнять «черную» работу: все хорошо, потому что смысл первичных ключей не изменился. Вчера это был простой счетчик, и он останется таким спустя годы при любых изменениях в бизнес-логике! Я удивляюсь, почему на форумах разработчиков ведутся дискуссии о том, какие первичные ключи применять: естественные или суррогатные. Нам всегда следует думать о наихудшем случае, и выше я описал, что может случиться с естественными ключами – головная боль и высокая вероятность багов. А что случится в самом худшем случае при использовании суррогатных ключей? В таблице будет одно дополнительное поле. Это не страшно – у нас нет цели сэкономить дисковое пространство: в наши дни мы можем оставить в ресторане сумму, превышающую стоимость жесткого диска.

Операторы SELECT

Метод «разделяй и властвуй» хорошо работает в SQL-запросах. По возможности, переносите часть запроса в подзапрос (если это не снижает производительность)! С точки зрения читабельности кода, всегда лучше написать несколько простых операторов SELECT, нежели облечь всю начинку в одну нечитабельную кучу таблиц, полей, операторов и аргументов. SQL производит впечатление легкого языка. Достаточно выучить 15-20 ключевых слов, чтобы сказать «Я его знаю!», но на самом деле это самый трудный язык. Не думайте об отступах в SQL-редакторе – эта проблема большей частью существует в языках с потоком «команда-после-команды», а не в SQL-языке четвертого поколения. В нем есть другая сложность: порой вы видите тяжелый запрос и хотите сменить профессию… Так почему бы вместо него не написать несколько более простых? Давайте сравним два подхода в простом запросе, который выдает название страны по городу, присутствующему в адресе:

*** ПЛОХОЙ код («все в одной куче»): ***

```
SELECT @country_name = country.country_name
  FROM country,
       city,
       address
 WHERE country.country_id = city.country_id
   AND city.city_id = address.city_id
   AND address.address_id = @address_id;
```

*** ХОРОШИЙ код («разделяй и властвуй»): ***

```
SELECT @country_name = country_name
  FROM country
 WHERE country_id =

       (SELECT country_id
          FROM city
         WHERE city_id =

               (SELECT  city_id
                  FROM address
                 WHERE address_id = @address_id))
```

Во втором варианте (по сравнению с первым) вы точно знаете, где начать исследование – в самом глубоком запросе (с другой стороны, если у вас почасовая оплата, вам больше подойдет метод «кучи»…)! ОК, он возвращает address_id и затем преобразует запрос следующего уровня в что-то вроде *«SELECT city_id FROM address WHERE address_id = 12345"*. И так далее. Как видите, этот пример очень легкий. Разбивка его на подзапросы не слишком помогла нам стать счастливее, но что вы скажете о реальных, повседневных SQL-запросах? К сожалению, иногда первый метод – единственно возможный…

Также существует третий метод: каждый раз выполнять «SELECT INTO переменная» и передавать эту переменную в следующий запрос как аргумент извлечения. Это наилучшее решение с точки зрения читабельности:

*** ОЧЕНЬ ХОРОШИЙ код («разделяй и властвуй» + переменные): ***

```
SELECT @city_id = city_id
  FROM address
 WHERE address_id = @address_id

SELECT @country_id = country_id
  FROM city
 WHERE city_id = @city_id -- выбор по ранее заданной переменной

SELECT @country_name = country_name
  FROM country
 WHERE country_id = @country_id -- выбор по ранее заданной переменной
```

Преимущества этого подхода:

- Ясность представления – убедитесь, как легко видеть происходящее!
- Если не получен ожидаемый результат запроса, в отладчике можно увидеть, на каком шаге произошла неудача (в предыдущих двух случаях можно увидеть только окончательный результат)

Но также существуют маленькие недостатки:

- Если SQL-запрос внедрен в скрипт клиентского приложения, он ухудшит его производительность, поскольку приложение чаще обращается к серверу баз данных и получает с него данные, а также код успеха/неудачи операции. Но не забывайте, что в большинстве случаев человек не почувствует разницы в миллисекунды. В наши дни самый дорогой ресурс – это время программиста, а не машинное время, как в 60-е годы.
- Этот метод неприменим, если SQL представляет собой строковое свойство объекта, который извлекает данные (например, SQL-строка, передаваемая в SqlDataAdapter в ADO.NET).

Но перемещение SQL-запросов в хранимую процедуру, ее вызов из клиента и использование как источника данных объекта позволит легко преодолеть эти две проблемы. Фактически, это должно выполняться путём создания клиент-серверных и распределённых приложений. Об этом ниже.

Выражения WHERE

Тем не менее, если вы применяете подход «все в одной куче» (например, из соображений производительности – некоторые СУБД недостаточно умны, и разбивка на подзапросы может привести к полному сканированию таблиц, так что всегда проверяйте execution plan!), используйте «новый» (ANSI) синтаксис для объединения таблиц (с ключевым словом JOIN) вместо перечисления таблиц в выражении FROM и ограничений в WHERE («старый» синтаксис). Конечно, если это допускается стандартами кодирования, принятыми в вашей компании. Старый подход я применял несколько лет, пока не оказался в проекте, где понадобилось принять правила кодирования, которые заставляли разработчиков использовать «новый» метод. Первые дни мне было немного дискомфортно, но теперь я счастлив и всем моим друзьям-разработчикам рекомендую перейти на стандарт ANSI. Он более понятен. Когда вы взираете на одну таблицу в сложном join, вы действительно сфокусированы на этой таблице, поэтому SQL-выражения больше не навалены в одну кучу…

*** ПЛОХОЙ код: ***

```
SELECT @country_name = country.country_name
  FROM country,
       city,
```

```
        address
 WHERE country.country_id = city.country_id
   AND city.city_id = address.city_id
   AND address.address_id = @address_id;
```

*** ХОРОШИЙ код: ***

```
SELECT @country_name = country.country_name
  FROM country
  JOIN city
    ON city.country_id = country.country_id
  JOIN address
    ON address.city_id = city.city_id
   AND address.address_id = @address_id;
```

OUTER JOIN: только LEFT

Применяя OUTER JOIN, используйте только вариант LEFT во всем приложении (и даже во всей вашей карьере, если это возможно) – и никогда RIGHT! Это значительно упростит представление в воображении объединяемых таблиц: левая воображаемая таблица всегда имеет значения во всех строках, а в правой имеются «дыры» из NULLs… Или наоборот, вы всегда может применять RIGHT и никогда LEFT. Это не имеет значения, важно вот что: используйте только одно из них (собственно, никакой политики!).

INSERT: упоминайте поля

Только для SQL Server:

В выражениях INSERT пишите имя поля рядом со вставляемым значением («field_name = inserted_value»):

*** ПЛОХОЙ код: ***

```
    INSERT #recordset (
           company_number
          ,entered_date
          ,description
          ,accounting_flag
          ,interface_flag
          ,subcontractor_contact_ba_id)
    SELECT @company_number
          ,received_date
          ,full_description
          ,'Y'
          ,'N'
          ,NULL /* будет заполнено позже */
      FROM v_change_order_header
     WHERE ...
```

*** ХОРОШИЙ код: ***

```
    INSERT #recordset (
           company_number
          ,entered_date
          ,description
          ,accounting_flag
          ,interface_flag
          ,subcontractor_contact_ba_id)
    SELECT company_number = @company_number
          ,entered_date = received_date
          ,description = full_description
          ,accounting_flag = 'Y'
          ,interface_flag = 'N'
          ,subcontractor_contact_ba_id = NULL /* будет заполнено позже */
      FROM v_change_order_header
     WHERE ...
```

Это позволит вам сразу увидеть, какое значение идет в какое поле. Очень полезно при добавлении/удалении полей в/из оператор(а) INSERT с большим количеством полей. Не забывайте, что имя поля, написанное со знаком «=» перед вставляемым значением, ведет себя как комментарий: вставка в любом случае происходит по позиции поля в части INSERT. Таким образом, если поле «last_name» указано третьим в части INSERT, и вы пишите «first_name = @first_name» в третьей строке части SELECT, вы заполните поле «last_name» значением, хранимым в переменной @first_name! К сожалению, это невозможно в Oracle, но для такого же эффекта можно задействовать комментарии («inserted_value /* field_name */» вместо «field_name = inserted_value»).

Кстати, касаясь оператора INSERT… Если вы не вставляете в поле никакое значение, вы просто можете пропустить это поле в выражении. Но более красиво явным образом вставить NULL (как в поле ‘subcontractor_contact_ba_id’ в примере

выше). Почему? Во-первых, разработчики увидят, что существующие колонки не заполнены оператором INSERT (иначе, они сочтут, что таблица меньше, чем на самом деле). Во-вторых, это своего рода комментарий, говорящий о том, что разработчик не забыл заполнить поле и намеренно оставил его пустым. Это особенно хорошо при начальном INSERT во временную таблицу в вашей хранимой процедуре: поля, оставленные пустыми, будут впоследствии заполнены в процедуре с помощью UPDATE временной таблицы (в таких случаях я даже добавляю комментарий: «/* will be populated later */»).

Временная таблица в хранимой процедуре

Если во временной процедуре для формирования возвращаемого набора записей (recordset) применяется временная таблица, будут полезны следующие правила:

- Поля в операторах «CREATE TABLE #recordset…», «INSERT #recordset…» и финальном аккорде вашей симфонии – «SELECT … FROM #recordset» – должны быть перечислены в одном и том же порядке. Не забывайте об этом правиле при добавлении поля к набору записей процедуры. Одна из главных концепций реляционных баз данных – «доступ к полям по их именам, а не по местоположению», но, пожалуйста, не следуйте этому правилу в данной ситуации. Мы говорим об упрощении нашей работы! :-)
- Иногда во временных таблицах есть дополнительные поля которые не возвращаются в наборе данных, но помогают заполнить возвращаемые поля. Например, с помощью оператора «UPDATE #recordset…», следующим за основным «INSERT #recordset…». Нет ничего плохого в пометке двух групп полей соответствующим комментарием (что-нибудь вроде «Поля, которые будут возвращаться в наборе данных» и «Поля для внутреннего использования»):

```
CREATE TABLE #recordset (
     /*********** Поля, которые будут возвращаться в наборе данных: ***********/
     start_date             datetime         NULL
    ,end_date               datetime         NULL
    ,certificate            varchar(200)     NULL
    ,company                varchar(200)     NULL
    ,date_taken             datetime         NULL
    ,comments               varchar(200)     NULL
    ,class_description      varchar(100) NOT NULL
    ,class_code             varchar(50)      NULL
    ,recertification_date   datetime         NULL /* recertification_period, прошедший после date_taken */
     /*********** Поля для внутреннего использования: ***********/
    ,student_id             integer      NOT NULL
    ,sc_id                  integer          NULL
    ,recertification_period tinyint          NULL /* необходимо для вычисления recertification_date */)
```

Преимущества:

- Легче обеспечить заполняемость всех возвращаемых полей;
- Наличие информации о том, что собой представляет набор записей процедуры – включая типы данных и длину полей (взглянув на финальный SELECT в конце процедуры, вы увидите только имена полей и порядок, но не типы данных и длину).

Проверка наличия сущности

Элегантное программирование подразумевает не только создание «косметически элегантного» и легко читаемого кода. Например, неэффективный код не выглядит слишком элегантным даже если он косметически совершенен… Для меня некомфортна следующая конструкция для проверки наличия сущности (entity):

```
SELECT COUNT(*)
  INTO counter
  FROM...;
IF counter > 0 THEN...
```

Я не понимаю, почему разработчики выполняют полное сканирование таблицы (зачастую это подталкивает пользователей к удовольствию минутами наблюдать за переворачивающимися песочными часами). Существует другой, более подходящий способ, который немедленно сообщит о наличии сущности после первого ее обнаружения в таблице (которая может быть очень большой):

```
BEGIN
   SELECT 1
     INTO existence_flag
     FROM DUAL
    WHERE EXISTS (SELECT 1 FROM...);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      existence_flag := 0;
END;
IF existence_flag = 1 THEN...
```

Слой «Страж Данных»

Иногда мы видим SQL-выражения в клиентском приложении или среднем звене N-tier приложения. Это не делает нас счастливыми. Гораздо лучше, если SQL вообще отсутствует в уровне бизнес-логики (Business Logic Layer, BLL) и даже в уровне доступа к данным (Data Access Layer, DAL). При создании нового клиент-серверного или распределенного приложения, ориентированного на работу с данными, есть возможность существенно облегчить свою жизнь, подумав о создании Слоя «Стража Данных» (Data Sentinel Layer). Это мое название, официально такого термина нет. Представленный слой обеспечивает взаимодействие между приложением и данными, хранимыми в базе. На самом деле, это набор хранимых процедур, который действует как API между DAL клиентского или среднезвенного приложения, написанного в среде .NET, Java, PowerBuilder и т.п., и хранилищем данных (таблицы баз данных). Права на доступ и управление данными напрямую есть только у этих процедур. DAL может лишь вызывать процедуры, которые служат основными «рабочими лошадками» такой системы. Таким образом, SQL-операторы будут отсутствовать в клиентском/среднезвенном приложениии – и даже в DAL. Да, да, я говорю серьезно – большое приложение уровня предприятия, ориентированное на работу с базой данных, может прекрасно существовать без SQL за пределами базы!

Значит, если объект в классе DbCommand среды ADO.NET применяется в DAL, задайте его свойство CommandType как StoredProcedure, а свойства SelectCommand, InsertCommand, UpdateCommand и DeleteCommand заполните именами хранимых процедур для SELECT, INSERT, UPDATE и DELETE, а не жестко определенными SQL-операторами как в случае определения CommandType как Text.

Обращали ли вы внимание на череду замещения одних инструментов разработки другими за последние двадцать-тридцать лет? COBOL, C, C++, PowerBuilder, Java, нынешний золотой век .NET… Никто не знает, что индустрия предложит завтра! Если же вы применяете описанный подход и программируете бизнес-логику в хранимых процедурах (а не в BLL вашего приложения, привязанного к нынешней технологии), то в будущем у вас будет меньше трудностей с переводом системы на новые технологии. Я уверен, что Oracle и SQL Server с их хранимыми процедурами будут существовать даже после конца света!

Итак, вперед!

- В *Oracle* все процедуры, обслуживающие одну таблицу, должны быть объединены в один пакет. Например, для таблицы Order его можно назвать pckOrder. Хранимые процедуры следует именовать единообразно во всех пакетах, например, Sel, Ins, Upd и Del. Конечно, пакет может также содержать другие процедуры и функционировать, скажем, как SelOpen (для извлечения заказов в статусе ‘OPEN’), GetLastOrderDateForCustomer и т.п. Важно придерживаться следующего соглашений об именах: «Sel» подразумевает возврат набора данных с помощью REF_CURSOR, имитирующий возврат данных SQL-оператором SELECT, а «Get» означает возврат скалярного значения (или значений) через оператор RETURN или параметр(ы) OUT (как классический метод Get…).
- В *SQL Server* названия всех процедур, обслуживающих одну таблицу, должны начинаться с одинакового имени сущности или префикса. Например, для таблицы Order: OrderSel, OrderIns, OrderUpd, OrderDel (для основных операций с БД) и OrderSelOpen, OrderGetLastOrderDateForCustomer (для дополнительных операций). Важно придерживаться следующего соглашения об именах: «Sel» подразумевает возврат набора данных оператором SELECT в конце хранимой процедуры, а «Get» означает возврат скалярного значения (или значений) через параметр(ы) OUTPUT.
- Полностью откажитесь от триггеров БД и не создавайте бизнес-логику, которая автоматически выполняется при вставке/обновлении/удалении записей (скажем, при проверках и обновлениях других таблиц) в BLL. Все автоматические действия, обычно программируемые в триггерах и BLL, следует переместить в хранимые процедуры, используемые для вставки, обновления и удаления записей. Такой подход несет громадное преимущество: все автоматические операции, относящиеся к таблице, оказываются В ОДНОМ МЕСТЕ. Взглянув на процедуру, скажем, вставки записи, вы увидите ПОЛНУЮ КАРТИНУ ОПЕРАЦИИ В БД, а не ее часть. И вам не нужно будет беспокоиться о том, что какие-то операции будет написаны дважды, или другой код затрёт/переопределит результаты ваших изменений.
- Все хранимые процедуры должны возвращать понятное сообщение об ошибке (через оператор RETURN или текстовый параметр OUT/OUTPUT – выберите один вариант и придерживайтесь его), которое готово к отображению в окне сообщений. NULL возвращается в случае удачного выполнения. Таким образом, вызывающее приложение должно всегда проверять код возврата.
- Все хранимые процедуры, применяемые для операций SELECT, INSERT, UPDATE и DELETE, должны возвращать количество затронутых записей (через оператор RETURN или числовой параметр OUT/OUTPUT). Таким образом, вызывающее приложение может проверить это количество без дополнительного обращения к БД.

4. Соображения по разработке в целом

Помечайте нерешенные проблемы

Если в скрипте есть неполадки, и вы не хотите исправлять их немедленно (например, заняты чем-то другим или улетаете на Гавайские острова через 3 часа), выделите это место вашей личной строкой-пометкой (конечно, в виде

комментария). Например, моя личная метка – «???mz» (инициалы для избежания коллизий с другими разработчиками). И не забудьте описать проблему парой слов – помогите себе будущему! Конечно, вы забудете о помеченных местах (особенно, после Гавайских островов), но Глобальный Поиск по вашей метке приведет вас прямо к забытым проблемам. Выполните этот поиск после завершения разработки, когда вы собираетесь сделать check-in. Это позволит избежать забытых «дыр».

Файлы-помощники

Следующие строки не связаны с элегантным программированием, но я решил разместить их здесь, поскольку они лежат в рамках предназначения статьи – облегчить жизнь программистов. Как правило, наша работа состоит из задач протяженностью в часы и месяцы (обычно несколько дней или недель). Для каждой задачи я создаю папку на сервере в корпоративной сети, на котором ежедневно выполняется резервное копирование. Эта папка будет содержать все материалы, относящиеся к задаче. Названия папок состоят из двух частей: число (начинается с 001 для правильной сортировки по времени) и описания (позволит впоследствии найти нужную папку). Вот примеры: «002 Training Views – change source tables», «004 Log Reports». Каждая такая папка содержит (стандарт):

- Подпапка «SQL» для скриптов, запускаемых по БД (CREATE, INSERT и т.д.)
- Файл «_obj.txt», в который записывается каждый объект после операции check-out. Этот файл разделен на маленькие части для каждой библиотеки (или схемы – для объектов БД). У каждого объекта своя строка, состоящая из двух колонок: имя объекта и краткое описание изменений, выполненных в ходе работы над задачей. Позже при операции check-in я копирую это описание в поле «Comments» системы управления версиями. Структура файлов выглядит так:

```
AAA_library:
        class_1                 Description of what is made in class_1
        class_2                 Description of what is made in class_2

BBB_library:
        class_3                 Description of what is made in class_3
        class_4                 Description of what is made in class_4

XXX_DB_schema:
        db_object_1             Description of what is made in db_object_1
        db_object_2             Description of what is made in db_object_2

YYY_DB_schema:
        db_object_3             Description of what is made in db_object_3
        db_object_4             Description of what is made in db_object_4
```

  Если я выполняю check-in всех классов, а затем check-out некоторых из них (например, для работы над новым релизом или исправления бага), я открываю новую секцию в этом файле. Почему? Потому что это «другая история» (с точки зрения процесса check-in/check-out). Наличие такого файла особенно полезно при одновременных check-out-операциях над классами для различных задач и смешивании их в одной библиотеке (например, PBL в PowerBuilder). Когда мне нужно выполнить check-in классов только для одной задачи, я просто заглядываю в файл без опасений забыть классы или выполнить check-in классов, принадлежащих другой задаче.

- Файл «_misc.txt» для заметок – обычно их много.
- Ярлык к файлу «_todo.txt, куда записываются все отложенные задачи, которые нужно выполнить в будущем (например, «Когда Билл Гейтс выполнит check-in в классе BlahBlahBlah, найти «???mz» в его методе BlahBlahBlah и…»). У меня один такой файл, служащий для различных задач (при начале каждой задачи копируется только ярлык). Благодаря этому, я вижу все планируемые изменения в одном месте. Вариант создавать todo-файлы для каждой задачи не сработал: я никогда не открывал их после завершения задачи!

И, конечно же, в каждой папке могут быть и другие файлы. В структуре папки они располагаются после описанных выше файлов, у которых имена начинаются с символа подчеркивания.

Документ по сценариям модульных тестов

При создании каждого блока приложения (скажем, новый экран или процесс) создавайте Word-документ «Сценарии модульных тестов» («Unit Test Cases»), который, по сути, представляет собой таблицу с пошаговыми инструкциями: что нужно сделать и какие результаты следует получить (колонка «OK» служит для пометок «успех/неудача» на соответствующем шаге тестирования).

Вот пример такой таблицы:

Тестируемый элемент	Выполняемые шаги	Ожидаемые результаты	OK?
Окно AAA	# Открыть окно AAA (меню «Open >…..”). # Вставить запись в объект данных и сохранить.	# Поле «bububu» задействовано и	–

	# Закрыть окно AAA. # Открыть окно BBB (меню «Open >…..”). # Добавить новую запись в объект данных «…».	содержит фокус. # Выпадающий список поля «bububu» содержит значение, вставленное в окне AAA.	
Удаление записи	# В выпадающем списке поля «bububu» выбрать значение, вставленное в окне AAA, и сохранить. # Закрыть окно BBB. # Открыть окно AAA. # Попытка удалить ранее вставленную запись.	# Невозможно: появляется сообщение.	–
–	# Закрыть окно AAA. # Открыть окно BBB.	# Поле «bububu» в DW «dw1″ заблокировано.	–
–	# Удалить из объекта данных ранее вставленную запись «…». # Сохранить. # Закрыть окно BBB. # Открыть окно AAA. # Попытка удалить ранее вставленную запись.	# Запись удалена, не выводится никаких сообщений.	–
Сохранить	# Сохранить. # Закрыть окно. # Снова открыть окно. # Найти ранее сохраненную запись.	# Появляются все сохраненные значения.	–

Колонка «Тестируемый элемент» дает краткое описание функциональности, тестируемой в строке. Если тест представлен более чем одной строкой, то достаточно привести описание один раз и оставить это поле пустым в последующих строках (в примере выше 3 строки описывают тестирование удаления записи, но «Удаление записи» присутствует только в первой строке из трех).

Важно! Поле «Ожидаемые результаты» должно хранить только результаты, интересные с точки зрения текущего теста! Если в результате ваших действий получен результат, который не является предметом текущего теста, то опишите этот результат в колонке «Выполняемые шаги». Например, если кнопка «Cars» открывает окно «Cars» и так происходит много лет (и сейчас не тестируется!), напишите:

```
# Нажать кнопку "Cars" - откроется окно "Cars".
# Нажать кнопку "Add"...
```

Или так:

```
# Нажать кнопку "Cars".
# В открывшемся окне "Cars" кликнуть на кнопку "Add"...
```

Но если вы только что создали окно «Cars» и тестируете, как оно открывается, то напишите в колонке «Шаги»:

```
# Нажать кнопку "Cars".
```

И в колонке «Ожидаемый результат»:

```
# Окно "Cars" открылось.
```

На создание такой таблицы потребуется время, но оно окупится! Такие документы являются неотъемлемой частью проекта (точно так же как и код на языке программирования). Вы же не скажете «Нет, я не буду писать код, потому что это займет время!», верно? Если в вашей компании подготовка «Инструкций по модульным тестам» не является общепринятой практикой, поделитесь идеей со своим менеджером!

Расскажите о следующих преимуществах:

- Работа тестировщика будет проще и быстрее, без догадок и туманных предположений о работе модуля.
- Программист сможет восстановить в памяти мельчайшие детали разработки, выполненные в далеком прошлом.
- Иногда создание такого документа высвечивает проблемы и привносит хорошие идеи в разработку (сейчас вы сможете сконцентрироваться на том, ЧТО нужно сделать, а не КАК это технически реализовать).
- Написание такого документа ПЕРЕД началом разработки (и создание модуля приложения ПО документу) позволит увидеть и решить проблемы на раннем этапе. Легче переписать Word-документ, чем переделать уже созданный и частично протестированный модуль. Поэтому напишите документ, отдайте его сотрудникам бизнес-отдела, у которых вы до этого собрали информацию по разрабатываемому модулю, и попросите их представить, будто они тестируют уже созданный блок приложения. Если они захотят изменить/добавить/убрать что-либо, возвращайтесь к документу MS Word, внесите соответствующие изменения и вернитесь к «бизнесменам». Выполняйте этот алгоритм в цикле (воспользуйтесь лестницей, если эти ребята сидят на другом этаже – очень полезно для здоровья программистов, занятых на сидячей работе!). Продолжайте

процесс до получения окончательной версии документа и только после этого приступайте к реализации его в коде.
- Модульные тесты, время от времени выполняемые по таким документам, помогут убедиться в том, что модуль не задет изменениями в других модулях или добавлениями новых («регрессионное тестирование»).

Одним словом, такой подход избавит вас от головной боли и сбережет деньги вашей компании.

EOF :-)

Оригинальная публикация: Elegant Coding Habits.

Рубрика: Методики
Метки: CSharp, SQL
Комментарии (10)
Отправить в Twitter, Facebook, ВКонтакте, Google+

Комментарии (10)

[...] публикация: Elegant Coding Habits. Продолжение следует. Рубрика: Методики Метки: CSharp, SQL Комментарии [...]

Навыки элегантного программирования. Часть 1. « OpenQuality.ru | Качество программного обеспечения | Опыт экспертов | 26.09.2010 | 8:26 пп. Ответить #

Интересно и по делу.
Автору благодарность за несение достойных идей в массы.

Денис | 14.10.2010 | 4:32 пп. Ответить #

О да. Я плакал. Я ошибся насчет буржуйских кодеров в кометтарии к первой статье. Автор сам с дерева еще не слез, а уже учит как «надо» делать. Первый же пример «хорошего» SQL — двойное декартово произведение одного множества. Я в шоке. Зачем? Массив данных для обрабртки растет по экспоненте на ровном месте. Не, я понимаю, конечно, что если в таблице 10 записей, то обработка 1000 записей вместо 10 не свалит sql-сервер, а всего в 100 раз увеличит время обработки запроса, но рекомендовать такое в качестве примера хорошего кода… Да пример плохого кода из этого же примера лучше примерно в 100 раз. Как минимум по производительности. В общем автора в школу, пусть математику учит.
Переводчику советую давать читать переводы специалистам перед публикацией. Публиковать статьи идиотов, которые учат как писать условия и циклы (детский сад?), называтьПеременныеОченьОченьДлинными ИменамиКоторыеНеПомещаютсяВОднуСтроку вместо использования контекста применения, писать SQL-запросы, которые повалят любую базу, и вообще переводить/публиковать авторов, которые несут полную чушь, или рассказзывают очевидные вещи (ПРИВЕТ КО) — неуважение к собственному труду и своей аудитории. А не дай бог юниор какой недоучившийся прочитает такое и решит что это правда? Это же интелектуальный калека сразу. Его потом пристрелить легче, чем поломаный мозг исправить.

Alex Sander | 27.02.2014 | 3:13 пп. Ответить #

Alex,

Я надеюсь, что Мартин Фаулер уже слез с дерева. В своей книге по рефакторингу он четко разделяет рефакторинг с целью улучшения читабельности кода и его правки в будущем, с одной стороны, и оптимизацию производительности, с другой. Он говорит о том, что на этапе рефакторинга не надо думать о производительности – к этому вопросу можно вернуться, если проблемы с производительностью действительно возникнут. Более того, тему производительности не обошел и автор статьи: «По возможности, переносите часть запроса в подзапрос (если это не снижает производительность)!»