Проектирование СУБД

А. Усов, 1999 (компиляция из писем в конференции fido.su.dbms)

1

Проектирование сложных систем сталкивается с большим числом проблем, большинство из которых не имеет простых решений. Те, кому приходилось создавать и сопровождать системы автоматизации управленческой деятельности предприятий или крупных подразделений, представляют, что означает работать с огромным числом таблиц, триггеров, индексов, представлений (view) и хранимых процедур. Сложность связей усугубляется сложностью логики предметной области. Не всегда очевидно, что необходимо выполнять в БД, а что на иных уровнях реализации. Можно только приветствовать развитие клиент-серверной технологии, и появление многоуровневых систем. Но при этом необходимо заметить, что признанной методологии разделения системы на несколько уровней не существует. Здесь можно добавить, что разработка сложных систем, использующих СУБД, имеет ту же природу (логику), что и разработка других сложных систем, например тех же СУБД.

Появление и распространение технологии объектного проектирования вызвало определённый ажиотаж, вокруг этой технологии. Сегодня можно столкнуться с самым широким диапазоном мнений относительно применимости этой технологии при разработке сложных систем. Стоящие на одном полюсе, доказывают, что без внедрения объектной технологии проектирования невозможно создавать системы требуемого уровня сложности. Оппоненты ехидно замечают, что пока главное достижение объектной технологии состоит в том, что она значительно увеличила... размеры программ. Но и среди сторонников объектного подхода нет единодушия. Если вы знакомы с работами Г. Буча, Шлейра и Мейлора, Йордана, Страупстрапа и других представителей этого лагеря, то наверняка обратили внимание, что их подходы к проектированию достаточно сложно назвать даже похожими. Хотя нельзя не отметить, что постоянно делаются попытки сблизить позиции. Тем не менее, говорить о единой платформе, единой методологии при проектировании систем, пока преждевременно.

Если нет возражений со стороны модератора и тех, кто читает материалы данной конференции, то мне бы хотелось обсудить здесь вопросы проектирования сложных систем. Первоначально я бы изложил основные положения, а затем проиллюстрировал их примерами. В качестве одного из этих примеров предлагается собственно проект СУБД, а в качестве другого примера проект системы управления предприятием. Если есть иные предложения и/или пожелания (вплоть до запрета публикации материалов), то лучше их высказать сейчас. Надеюсь, что всем понятно, разработка сложной системы требует достаточно больших сил на одно только описание предметной области.

(Все материалы будут изложены на основе моей статьи, посвящённой тем же вопросам. Однако статья содержит более подробное и детальное описание)

2

Проектирование сложных систем очень часто начинается с длительных и, как правило, безуспешных попыток формализации, достаточно подробного описания предметной области. Составлению спецификаций сложных систем посвящено множество исследований, книг, статей и диссертаций. Но... Исходно неверные предпосылки ведут к заведомо неправильным результатам. Детальное описание сложной системы - это фикция. Это положение проистекает, во-первых, из сложности самой системы, а, во-вторых, из предположения, что за время процесса формализации,

проектирования, разработки и сдачи в эксплуатацию сложная система не претерпит существенных изменений, способных низложить существо проекта. Другими словами, объективно существует противоречие между тем, что не предпроектной стадии необходимо выполнить формализацию того состояния системы, в котором она может оказаться через сколь угодно большой промежуток времени своего существования. Можно ли описать неведомое? Предположим, что мы приступаем к проектированию СУБД, но при этом знаем, что и платформа ("железо", ОС) и требования к системе могут коренным образом измениться в любой момент. Как приступить к такому проекту? Изменения же, которые происходят на предприятиях, ещё более кардинальны. Стоит ли браться за разработку системы управления предприятием?

Если посмотреть литературные источники, то они, как один, не рекомендуют приступать к разработке, не имея строгого технического задания. С другой стороны, если посмотреть в той же литературе, то техническое задание даже на совсем простые системы занимает сотни страниц. А если система сложнее, чем учебный пример? В своё время Дейкстра потратил много сил и лет на попытку вывести способы верификации программ. Результат остался неутешительным: с ростом объёма кода сложность доказательства правильности программы увеличивалась быстрее сложности самой программы. В результате доказательство потенциально могло содержать значительно больше ошибок, чем сама программа. Техническое описание сложной системы, тоже не застраховано от ошибок. Проект, выполненный на основе такого технического задания, и имеющий собственные огрехи, более чем сомнителен. Программы же, написанные на основе такого проекта, и тоже имеющие свои собственные ошибки... Как здесь не вспомнить второй закон Вейлера: "Если бы строители строили здания так же, как программисты пишут программы, первый залетевший дятел разрушил бы цивилизацию".

Очень хотелось верить, что объектная технология сможет решить эту проблему проблем, но пока это не так. Если Вы знакомы с работами в этой области, то очевидно согласитесь с этой точкой зрения. Однако всё совсем не так плохо. Проблема имеет решение и оно совсем не так сложно, но чтобы им можно было воспользоваться необходимо сформулировать базовые принципы:

1. Любая сложная система не может иметь полного формального описания;
2. Любая сложная система должна динамично развиваться, а, следовательно, она не может находится в завершённом состоянии;
3. Любая сложная система разложима в конечном итоге на простые компоненты;
4. Сложность системы зависит от сложности составляющих её компонент и связей между ними, тогда, упрощение, например, классификация компонент и упрощение связей между ними способно понизить сложность любой системы;
5. Единственная задача любой системы - это научиться управлять, то есть, манипулировать связями между компонентами, и обслуживать компоненты;
6. Умение создавать простые и эффективные компоненты и столь же простые связи между компонентами адекватно умению превращать сложную систему в простую.
7. "Делать сложным - просто, делать простым - сложно".

С п.1 надо смириться. "Нельзя объять необъятное" (К.Прутков)

Из п.2 следует, что не надо пытаться разработать всю систему целиком. Создание системы начинается с макетирование и заканчивается рабочим вариантом. Между этими стадиями нет никакой границы. Сегодняшний рабочий вариант возможно послужит всего лишь макетом для системы, которая потребуется завтра.

П.3 далеко не нов, он лежит, в том числе, в основе структурной декомпозиции.

П.4 определяет ведущую роль тех концепций и направлений, которые позволяют получать более простые и удобные компоненты. Это прежде всего относится к объектной технологии.

П.5 определяет очень важное положение, о котором предстоит говорить ещё очень много, а именно, о разработке схем взаимодействия между отдельными частями системы.

Остальные пункты не требуют комментария.

3

Поскольку далее речь пойдёт об объектных анализе и проектировании, наверное имеет смысл рассмотреть общие положения объектной технологии. Прежде всего, необходимо определить место объектной технологии и её взаимосвязь с другими подходами к разработке программного обеспечения (ПО).

В литературе крайне скупо пишут о происхождении основных положений, на которых базируется объектная технология. Безусловно, истоки этой технологии лежат в методах структурной разработки ПО. Одна из целей структурного подхода заключается в декомпозиции кода. Декомпозиция кода подразумевает разделение программы на ряд подпрограмм. Это позволяло снизить число связей в программе и повысить её эксплуатационную надёжность, включая более простое развитие и модификацию. В свою очередь, одни и те же подпрограммы стало возможным использовать в различных программах, что служило дополнительным фактором повышения надёжности программ и ускоряло их разработку. Подпрограммы стали объединять в библиотеки, как правило, по функциональному признаку. Библиотеки подпрограмм включались в компиляторы, операционные системы, а также они поставлялись как самостоятельные программные продукты.

Не смотря, на прогрессивность, библиотеки подпрограмм имели один существенный недостаток. Дело в том, что даже безупречный код может работать неправильно, если данные, которые он обрабатывает меняются случайным образом. Но данные, в отличие от подпрограмм, работающих с ними, располагались в программе. Любое неосторожное изменение данных могло непредсказуемым образом сказаться на результатах работы программы. Появилась реальная потребность сблизить данные и код. Это было решено в модульном программировании. Модуль, в отличие от библиотеки, мог содержать и данные, и код.

С другой стороны, разделение библиотек по функциональному признаку оставалось справедливым и при комплектовании модулей. Следуя в этом направлении было гораздо проще, быстрее и удобнее разрабатывать модули узкоориентированные на работу в какой-то небольшой области. Данные, необходимые для работы объединялись в структуру, с которой взаимодействовал набор подпрограмм. Такой модуль можно считать прообразом объекта.

Однако данные, располагаемые в модуле не были защищены от внешнего, по отношению к модулю, ПО. И, следовательно, говорить о корректности данных, обрабатываемых подпрограммой было преждевременно. Поэтому следующим шагом было разделение модуля на две или более части. Как правило, модуль имел интерфейсную зону, где располагались данные и подпрограммы, видимые извне. А также, имелась зона реализации, недоступная для внешнего ПО. Критические, для работы модуля, данные располагались в зоне реализации и не могли быть случайным образом изменены извне. Это безусловно повышало надёжность модулей.

Но иметь "ненадёжные" данные в зоне интерфейса тоже не сулило ничего хорошего. Постепенно данные из интерфейсной зоны начали смещаться в зону реализации, а для того, чтобы внешнее ПО, могло взаимодействовать с этими данными в интерфейсной зоне объявлялся набор подпрограмм. Говоря современным языком данные и часть подпрограмм инкапсулировались в модуль, а для взаимодействия с модулем объявлялись интерфейсные подпрограммы.

Дальнейшее развитие модульного программирования привело к появлению в модуле областей, отвечающих за инициализацию модуля при загрузке и деинициализацию при выгрузке. Что соответствует действиям конструктора и деструктора при работе с объектом.

Аналогично можно рассмотреть развитие ПО в направлении повышения уровня абстракции, что в объектной технологии выразилось в наследовании и полиморфизме. Эти направления будут рассмотрены дальше. А это письмо мне бы хотелось завершить выводом, что объектная технология является логичным развитием идей структурного, модульного и многих других направлений развития ПО. И говорить о какой-то особой роли или даже оппозиции объектной технологии к другим технологиям неправомочно. Правильнее будет утверждение, что объектная технология объединила различные направления и придала им совершенно новое качество.

4

Мне, наверное, надо извинится за столь долгое вступление, но дело в том, что все вопросы, рассмотренные ранее имеют прямое отношению к процессу проектирования. А потому, мне показалось логичным изложить их до того, как переходить к более сложным вопросам. В этом письме будут рассмотрены основные положения объектной технологии.

Обычно в основе объектной технологии рассматривают такие концепции как инкапсуляция, полиморфизм и наследование. Некоторые из исследователей расширяют этот список, так Гради Буч, например, рассматривает абстрагирование, инкапсуляция, модульность и иерархию, а также три дополнительных концепции: типизация, параллелизм и сохраняемость. Мне не хотелось бы здесь обсуждать правомочность такого базиса. Наверное правильнее будет кратко рассмотреть традиционные концепции.

Инкапсуляция

Инкапсуляцию рассматривают с двух позиций: объединение кода и данных в единое целое и защиту данных и части кода. Про объединение кода и данных ещё предстоит поговорить более обстоятельно, поэтому пока обсудим вопросы защищенности. Инкапсуляция кода и данных важнейший компонент при проектировании систем. Благодаря тому, что реализация класса неизвестна внешнему ПО, можно допустить, что реализация может быть любой. Если же структура класса или его внутренние свойства видны на каком-то уровне проектирования, то привязка к этим элементам не позволит изменить реализацию класса. Таким образом, инкапсуляция - это защита тайны реализации. Защищая реализацию, мы развязываем себе руки для любых модификаций класса при сохранении неизменного интерфейса. Следствием этого положения является возможность плавного перехода из стадии макетирования в стадию рабочего проекта. Удобство такого решения переоценить очень сложно.

Действительно, в сложных системах важное (если не сказать - основное) значение имеют связи между компонентами. Создание и отладка слаженной работы большого числа компонент, как правило, намного сложнее, разработки отдельного компонента. Но сегодня этого мало, статичные связи, установленные при проектировании уже не удовлетворяют современным требованиям. Связи должны устанавливаться динамически, а это многократно усложняет любой проект. Поэтому при проектировании очень важно максимально быстро пройти стадию разработки компонент, дабы приступить к моделированию и отлаживанию связей. Не исключено, что при отладке связей и других частей системы первоначальные требования к компонентам претерпят изменения.

В своём письме (Пр.2) я говорил, что спецификация сложной системы - это фикция. Более реальный путь разработки таких систем - это моделирование с постепенным переходом в стадию рабочей системы. Без инкапсуляции выполнить этот переход едва ли возможно. Мало того, любая

сложная система должна иметь возможность развиваться, что тоже практически невозможно реализовать, если отдельные компоненты этой системы взаимодействуют между собой произвольным образом. Поэтому не пренебрегайте инкапсуляцией! Не верьте в частичную инкапсуляцию, инкапсуляция либо есть (код и данные невидимы извне), либо её нет (код и данные видны на каком-то из уровней разработки). Все примеры, которые будут приведены, основаны на полной инкапсуляции. Возможно эти примеры помогут Вам составить представление о том, для чего нужна инкапсуляция и как ей правильно пользоваться. К сожалению из знания C++, Delphi и прочих гибридных языков не следует необходимость, тем более умение, применять инкапсуляцию.

Полиморфизм

Полиморфизм является важнейшим и ключевым понятием объектной технологии. В практике сложилась несколько упрощённое представление о полиморфизме, как части наследования. То есть, считается, что полиморфными могут быть свойства классов, находящихся в связи наследования. Подобное "упрощённое" представление об этой концепции приводит к очень сложным компонентам, из которых собирается система. Например, в книге Г. Буча "Объектно-ориентированный анализ и проектирование с примерами приложений на C++" приводится пример обосновывающий необходимость введения множественного наследования. В примере описываются классы цветов и фруктов-овощей. Фрукты-овощи имеют плоды, но некоторые цветы тоже имеют плоды. Но ближайший общий суперкласс у цветов и фруктов-овощей плодов иметь не может (ведь не все растения имеют плоды). Следуя логике Г. Буча, необходимо создать класс плоды, и использовать его для множественного наследования. Это позволит получить и цветы, имеющие плоды и фрукты-овощи с плодами. Остаётся непонятным как к иерархии растений пристроить класс "Плоды".

Решение проблемы лежит в иной плоскости. "Иметь плоды" - это полиморфное свойство. Этим полиморфным свойством может обладать любой класс в иерархии, если мы свяжем данное свойство с выбранным классом. Это и есть механизм полиморфизма. Очень часто можно встретиться с ситуацией, когда различные сущности обладают одним и тем же свойством, например, летать могут птицы, насекомые, рыбы и различные летательные аппараты. Означает ли это, что их общий суперкласс имеет это свойство? Нет, конечно. Другими словами, полиморфизм существует объективно вне иерархии наследования. И это свойство полиморфизма позволяет получать семантически простые и эффективные решения. В примере проекта СУБД это свойство будет использоваться повсеместно.

Наследование

Наследование позволяет вводить в рассмотрение очень высокие уровни абстракций. Абстракции очень важны при работе над сложными проектами. Задавая поведение абстрактных классов, мы тем самым не только разделяем работу, но и существенно облегчаем смысловое восприятие, что для сложных систем очень важно. Реализация конкретного класса существенно облегчается поскольку его поведение задаётся абстрактными классами, и остаётся уточнить только порядок обработки тех или иных сообщений.

Полиморфизм, в той нотации, что изложена выше, не отменяет наследования. Наследование определяет поведение класса, отвечает за передачу классообразующих свойств. Здесь полиморфизма недостаточно, поскольку он работает на уровне отдельного свойства класса, но не их совокупности. Тем более полиморфизм не в состоянии передать структуру класса.

Множественное наследование - это пример скверной формализации. Не смотря на то, что большинство наиболее видных исследователей сходятся во мнении, что множественное

наследование необходимо, реально это не так. Мной многократно разбиралось множество примеров, где по уверениям невозможно было обойтись без множественного наследования. Всегда находилось более красивое и эффективное решение, не использующее множественное наследование. Корни множественного наследования лежат в том, что делается попытка описать какую-либо сущность с самых различных позиций. Каждой из таких позиций придаётся своя иерархия. В этом случае, сущность, лежащая на пересечении ветвей, этих иерархий должна обладать свойствами, передающимися по различным иерархиям. Последовательно применяя правило локализации и используя полиморфизм, можно и нужно свести рассмотрение к одной иерархии, являющейся основной в данной предметной области или её части.

Наконец, заканчивая рассмотрение наследования, необходимо отметить, что и одиночное наследование многогранно. Пока мы рассматриваем какую-то ветвь иерархии наследования, всё получается элементарно. Но ни наследование, ни полиморфизм, ни инкапсуляция не решают одну из наиболее важных проблем - проблему агрегирования нескольких классов в единое целое. Этому важному инструменту анализа и проектирования посвящено следующее письмо.

5

В этом письме я надеюсь закончить краткое рассмотрение элементов объектной технологии. В нём будет рассмотрена агрегация - объединение в единое целое различных компонент. Во многих работах посвящённых объектному анализу и проектированию так или иначе задевается тема объединения различных компонент в единое целое. Однако мне не доводилось видеть работ, где бы эта тема рассматривалась сколько-нибудь полно.

Агрегация

Если взглянуть на любой класс, то он представляет собой объединение подпрограмм и данных. Иными словами, класс является агрегатом. Агрегат характеризуется функциональностью. Функциональность класса определяется тем набором свойств, которые имеет класс. Интерфейс класса образуют интерфейсные свойства (в более общем случае - совокупность сообщений, на которые данный класс может отвечать или которые данный класс может обрабатывать). Данные класса образуют интерфейс необходимый для сохранения данных между обращениями к подпрограммам-свойствам класса. В простейшем случае, структура данных класса отражает состояние класса.

Однако потребность в объединении не исчерпывается на уровне класса. Объединением нескольких классов в единое целое занимаются контейнеры. Контейнер объединяет несколько классов, но объединяя различные классы, контейнер является принципиально иной сущностью, нежели составляющие его классы. "Собор, есть нечто совсем иное, нежели просто нагромождение камней. Собор - это геометрия и архитектура. Не камни определяют собор, а, напротив, собор обогащает камни своим особым смыслом. Его камни облагорожены тем, что они камни собора. Самые разнообразные камни служат его единству." (Антуан де Сент-Экзюпери). Автомобиль нельзя отождествить ни с шасси, ни с двигателем, ни с корпусом, ни с другой составной частью. Автомобиль - это транспортное средство.

С появлением такой сущности как контейнер, мы получаем ещё две иерархии - иерархию вложений и иерархию наследования реализации. Например, тот же автомобиль состоит из различных элементов, которые сами могут быть контейнерами для других элементов, более низкого (элементарного) уровня. В то же время автомобиль может быть суперклассом для легкового и грузового автомобилей. Эти подклассы получают "в наследство" ту реализацию, которую мы определили на уровне их суперкласса, например, двухосное шасси. Но они имеют право переопределить данное свойство и ввести трёхосное шасси для тяжёлых грузовых

автомобилей. Переопределение реализации контейнера - суперкласса тождественно переопределению подклассами полиморфных свойств своего суперкласса.

В то же время, контейнер сам является классом. Его функциональность реализуется через вложенные классы, а его интерфейс образован схемами. Схема - это распределение исходного сообщения, пришедшего к контейнеру, на сообщения к вложенным классам. Сам контейнер, как правило, никакой работы не выполняет, его основная задача - служить носителем логики взаимодействия вложенных в него подклассов при обработке сообщений. Можно сказать иначе, контейнер - это диспетчер сообщений, брокер запросов. Схемы обработки сообщений могут быть простыми и сложными. На сколько мне удавалось наблюдать, с повышением логического уровня (контейнер контейнеров) интерфейс контейнера существенно упрощается, а его внутреннее устройство усложняется, следовательно, усложняется и логика распределения сообщений. Высокоуровневые контейнеры могут не иметь формализованной логики обработки сообщений. В этом случае допустимо (полезно) пользоваться эмпирическими знаниями. То есть, такой контейнер может содержать элементы экспертных систем (базу знаний, фактологическую базу, машину вывода) или нейронную сеть. Это является атрибутом систем искусственного интеллекта.

Не менее важно отметить то, что при обработке входного сообщения схема контейнера определяет последовательность работы вложенных классов. Рассылка сообщений по вложенным классам происходит последовательно или параллельно. Участки параллельной рассылки могут сменяться участками последовательной рассылки и наоборот. Определение логики работы схемы сводится к определению какие сообщения передаются вложенным классам и в какой последовательности. Здесь же Определяются участки параллельной и последовательной обработки. Иными словами, схема представима в виде графа обработки сообщения.

Схемы контейнера - это полиморфные свойства контейнера. Они могут совпадать или не совпадать с полиморфными свойствами вложенных классов. Они могут прямо проецироваться на родственные полиморфные свойства классов или иметь более сложное распределение на несколько свойств и/или классов. Понятно, что подобная гибкость достигается исключительно благодаря механизму полиморфизма, ибо, как отмечалось выше, контейнер - это принципиально иная сущность, отличная от вложенных в него классов.

С точки зрения взаимодействия между вложенными классами и контейнером можно выделить динамические и статические контейнеры. В динамическом контейнере все вложенные классы обладают некоторым множеством одинаковых (с точки зрения интерфейса) полиморфных свойств. Обращение к вложенным классам происходит только через этот общий набор свойств. Логика работы динамического контейнера не зависит от количества вложенных классов и это количество может меняться в очень широких пределах. Примером такого контейнера является массив элементов, список, муравейник (пока мы не наделяем какого-то муравья индивидуальной ролью) или ученики класса. На другом полюсе находятся статические контейнеры. В статическом контейнере роль каждого вложенного объекта уникальна. При распределении сообщений роли объектов играют в этих контейнерах ключевую роль. Так для автомобиля роль двигателя и шасси принципиально различна. Поэтому здесь нельзя произвольно изменять количество вложенных классов. Это, как минимум, потребует переопределения схем контейнера. Наконец, существуют контейнеры обладающие свойствами и динамического и статического контейнеров. Например, отдел. Все сотрудники отдела являются служащими и к ним применимы любые действия применимые ко всем служащим: приём на работу, увольнение, переводы, начисление зарплаты и т.п. Но роль каждого сотрудника в отделе уникальна. При поступлении задания оно распределяется по сотрудникам с учётом той роли, которую они играют в отделе.

Есть у меня ощущение, что я пропустил какие-то детали, но если что потом можно к ним вернуться. А на этом я хотел бы завершить этот краткий обзор по элементам объектной

технологии. В следующем письме начнётся рассмотрение проекта СУБД.

6

В предыдущих письмах я действительно упустил один важный момент, о котором необходимо рассказать. Речь идёт о локализации. Под локализацией понимается локализация любых возможных изменений любого логического уровня, так чтобы изменения, производимые на одном уровне, не влияли на другие уровни. В некотором смысле, локализация расширяет понятие инкапсуляции. Инкапсуляция защищает реализацию класса, в то время, как локализация защищает реализацию логического уровня.

Как отмечалось в предыдущем письме, посвящённом вопросам агрегации, контейнеры включают в себя объекты некоторых классов. Контейнеры нижнего логического уровня образованы элементарными классами, то есть классами, которые сами контейнерами не являются. В свою очередь, контейнеры нижнего логического уровня могут быть включены в контейнеры более высокого логического уровня. При этом на одном логическом уровне могут существовать различные по своему виду контейнеры.

Локализация определяет:

1. Классы, принадлежащие различным логическим уровням могут взаимодействовать между собой только через установленный интерфейс.
2. Объекты, расположенные на одном логическом уровне могут взаимодействовать между собой только в рамках контейнера-владельца и только через посредство контейнера-владельца.
3. Объекты, вложенные в один контейнер, не могут взаимодействовать с объектами, вложенными в другой контейнер. Они также не могут непосредственно взаимодействовать с другим контейнером.

Другими словами, объект "видит" только свой контейнер. О существовании каких-либо других объектов в системе он не подозревает. Взаимодействие между объектом и владельцем происходит только в рамках интерфейса. Как правило, достаточно однонаправленного взаимодействия: контейнер отдаёт распоряжение, а вложенный объект его исполняет. Однако вполне возможна ситуация существования активных объектов, которые могут посылать сообщения своему владельцу. Примером таких объектов могут служить различные элементы графического интерфейса (controls), которые оповещают владельца (например, диалоговое окно), о том что пользователь выполнил какое-то действие. Другим примером таких объектов, могут служить компоненты компьютера способные возбуждать прерывания.

Независимость логических уровней столь же необходима, как и инкапсуляция. Можно сказать, что локализация - это инкапсуляция контейнеров. Дело в том, что инкапсуляция контейнеров отличается от инкапсуляции элементарных классов. Главное отличие состоит в том, что контейнеры могут менять свою реализацию (типы и количество вложенных классов и схемы обработки сообщений) непосредственно во время работы (run-time), в то время, как элементарные классы создаются и модифицируются только во время разработки (design-time). Это принципиальное отличие. Например, control (элемент интерфейса) создаётся программистом в design-time, а диалоговое окно, содержащее controls, создаётся в run-time (и не обязательно программистом).

В таблице 1 приведено сопоставление элементарных классов и контейнеров.

Таблица 1

	Элементарные классы	**Контейнеры**

Интерфейс	Свойства	Схемы
Функциональность	Подпрограммы	Объекты
Создание/изменение	Design-time	Run-time

Следствием различий между контейнерами и элементарными классами является тот факт, что необходимо разработать специальный инструментарий для работы с контейнерами. Этот инструментарий должен позволять конструировать контейнеры и их схемы обработки сообщений. Элементарные классы и контейнеры не исчерпывают всех уровней абстракций, присущих сложным системам, но для рассмотрения, оговорённых ранее примеров, этого будет достаточно.

Однако я отвлёкся, а надо закончить с темой локализации. Разработка серьёзного проекта не выполняется в одиночку, поскольку слишком много рутинной работы. Но при работе в коллективе очень важно чётко разграничить полномочия и обязанности между всеми членами рабочей группы. Локализация позволяет разграничить уровни, а, следовательно, и разделить разработку на независимые части. Независимость уровней позволяет, в том числе, проводить независимо и их разработку. Как отмечалось ранее, разработка начинается с простой модели и может не заканчиваться сдачей готового проекта в эксплуатацию (ибо нет предела совершенству). И здесь так же важно то, что развитие каждого логического слоя происходит полностью независимо от развития других логических слоёв. И, что особенно важно, так это то, что разделение по логическим слоям - это прямое отражение иерархичности любой сложной системы.

7

В этом письме рассматривается проект разработки СУБД. Было слишком наивно полагать что в нескольких письмах можно рассмотреть столь серьёзный проект целиком. Но это и не потребуется. Важно рассмотреть основные проектные решения (выбор модели) и определить пути развития (совершенствования) проекта. Мало того, в начале я специально максимально упрощу представления о том, что собственно хочется получить в результате разработки проекта, подчёркивая тем самым мысль о том, что любая сложная система не может иметь на предпроектной стадии полного формального описания. Да, это, как Вы увидите, и не нужно.

Было бы очень хорошо, если бы Вы попробовали построить модель, так как это будет описано в этом и последующем письмах, особенно, если Вы сомневаетесь в моих выводах и заключениях. На самом деле, построение такой модели весьма интересное и, надеюсь, поучительное занятие. По крайней мере, когда, десять лет назад, мы проделали этот путь, то это было очень интересно и увлекательно (хотя, справедливости ради, надо отметить что, последовательность наших действий была несколько иной).

Итак, предположим, что мы хотим построить реляционную СУБД, но все наши познания в данной области исчерпываются знакомством с теми статьями Э. Кодда, где он обрисовал свою реляционную модель. Из этих статей следует, что информация хранится в базе данных, в виде отношений (таблиц). Таблицу можно рассматривать как совокупность строк (кортежей) или столбцов (атрибутов). Каждый атрибут характиризуется типом данных, которые в нём хранятся. Схема отношения (заголовок таблицы) образуется из произвольного набора типа данных, характеризующих некоторую сущность предметной области (или только часть сущности).

Структурное программирование утвердило понятие структуры данных, которая представима набором полей различного типа данных. Это достаточно хорошо соотносится со схемой отношения, которая тоже представляет собой набор полей различного типа данных. Возможно это сходство послужило одной из причин того, что реляционные СУБД предпочитают хранить данные

в таблицах по кортежам, а не по атрибутам. Но, следует заметить, что с точки зрения объектной технологии, такой способ хранения не очень удобен. Что будут представлять собой элементарные классы? Кортеж? Но мы при проектировании СУБД понятия не имеем о том, каковы будут схемы отношений баз данных, спроектированных под эту СУБД. Даже при очень ограниченном количестве типов данных мы получим бесконечно большое количество их комбинаций. Такой путь не сулит хорошего решения.

Попробуем рассуждать иначе. Атрибут, в отличие от кортежа, имеет одинаковый тип данных во всех полях. Количество типов данных относительно невелико. Мало того, любые типы данных достаточно просто представить в виде обычной иерархии классов (типов данных). Тогда в качестве элементарного класса будет выступать атрибут, то есть объект обслуживающий некоторое хранилище однотипных данных. Примерная иерархия классов атрибутов представлена на рис.1.

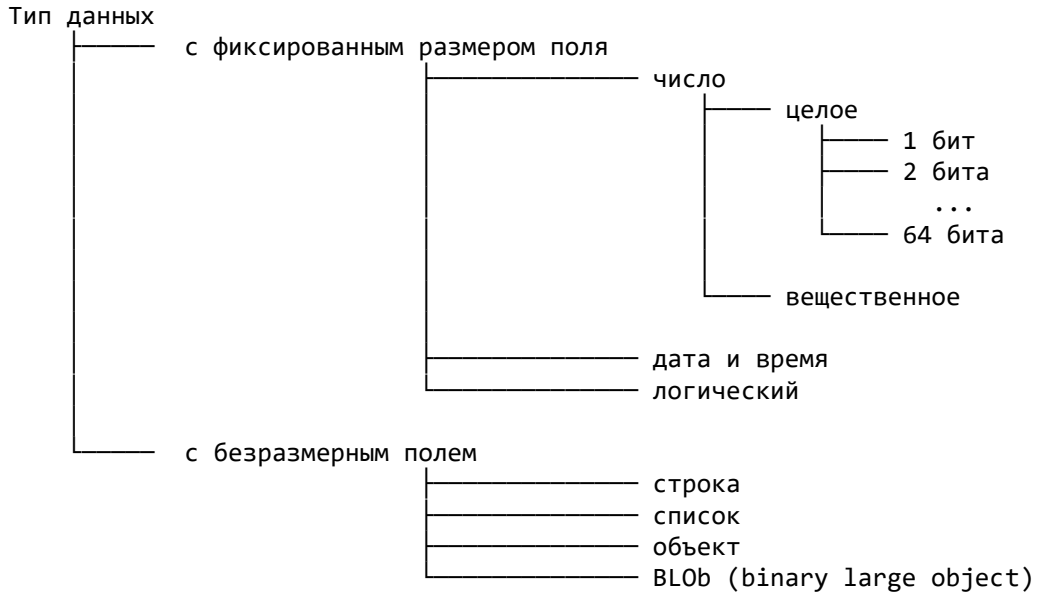


Рис.1 Примерная схема иерархии классов атрибутов

Данная иерархия не претендует на законченность. Но в неё достаточно просто уложить все типы данных, определённые в стандарте SQL 92. Эту иерархию достаточно просто пополнять новыми типами данных, если в них возникнет необходимость. Для моделирования достаточно реализовать всего несколько типов. И если остальная часть модели будет работоспособна, то всегда можно добавить недостающие типы данных.

Но сейчас интересно другое. Если атрибут является элементарным классом, то, в этом случае, отношение будет являться контейнером атрибутов. Но, тогда база данных - это контейнер отношений. Таким образом, мы получили три логических уровня. Каждый из этих уровней обладает независимостью от других. Так, например, отношение не интересует, как устроен тот или иной атрибут. Оно всегда обращается к атрибутам через определённый интерфейс (о нём пойдёт речь чуть позже). В свою очередь, база данных никак не связана с тем, как реализовано отношение. Реализация каждого уровня может быть изменена в любое время и это не скажется на логике работы других уровней. Каждый уровень может быть разработан отдельной группой разработчиков и их направления работы не будут пересекаться между собой.

Теперь можно предположить, что основа проекта заложена и можно переходить непосредственно к созданию модели. Об этом следующее письмо.

8

В начале моделирования необходимо определить тот интерфейс, который должны обеспечить элементарные классы. Так как все остальные логические уровни, основаны на данных классах. Ранее отмечалось, что основная работа происходит в элементарных классах, а, следовательно, применительно к СУБД можно отметить, что запросы приходящие от пользователя, так или иначе, но будут спроецированы на элементарные классы. В свою очередь, запросы пользователей можно представить следующей иерархией:

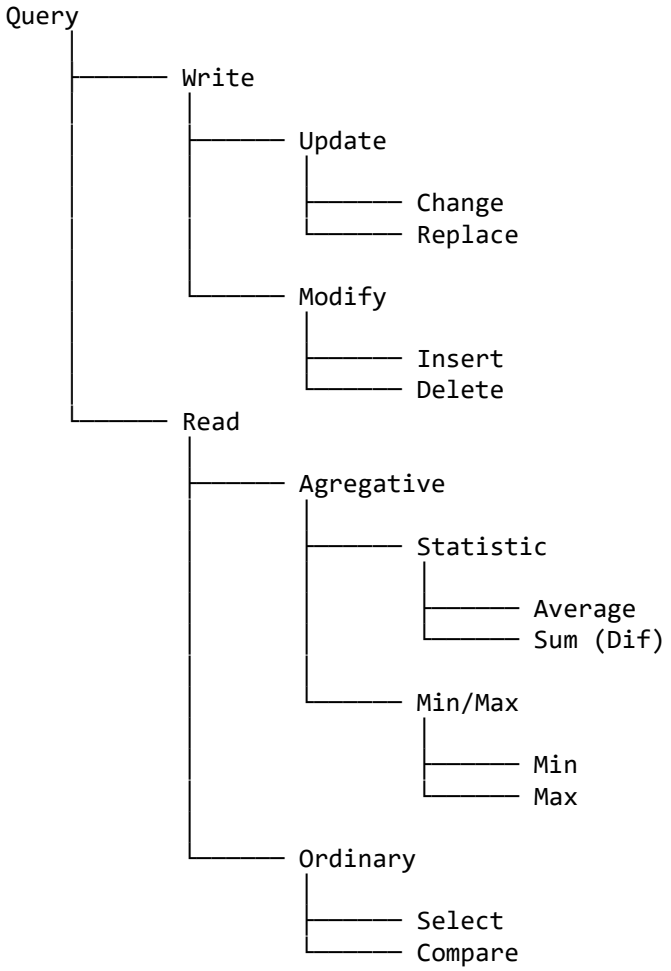


Рис. 2. Примерная иерархия типов запросов

Надо отметить, что представлять запросы в виде иерархии не обязательно, те более, не обязательно приводить запросы к бинарному дереву. Однако это очень удобно, например, для проверки прав доступа. Проверка будет сведена к банальной операции наложения битовых масок. Другим удобством иерархического представления является возможность лёгкого встраивания новых видов запросов, если они потребуются (речь идёт, прежде всего, об агрегатных функциях).

Собственно это бинарное дерево запросов и определяет тот набор свойств, который должен быть реализован у элементарных классов. Желательно расставить приоритеты, то есть определить, какие свойства являются наиболее необходимыми в первой реализации модели. Кроме того, не спешите реализовывать сложные формы хранения или сложные методы поиска и доступа. Сейчас важно построить с минимальными затратами модель и отладить связи между элементами этой модели.

Как отмечалось в предыдущем письме, нижний логический уровень образуется атрибутами.

Атрибут представляет собой набор однотипных данных. Самый простой способ реализации такого набора - это динамический одномерный массив. Для серьёзной работы с данными такой способ хранения не годится, но для моделирования этой работы, он вполне пригоден. Поэтому пока воспользуемся этим решением.

При рассмотрении иерархии атрибутов было рассмотрено два вида - атрибуты, имеющие фиксированный размер и атрибуты произвольного размера. Реализовать атрибут фиксированного размера в виде динамического одномерного массива просто.

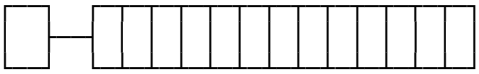


Рис. 3. Схема атрибута фиксированного размера

Теперь необходимо попытаться представить схему атрибута произвольного размера. Можно размещать элементы (значения) последовательно, разделяя их каким-то терминатором. Но подобная схема не позволит иметь прямой доступ к каждому элементу (значению), что является неприемлемым решением. Другой способ заключается в том, чтобы вынести относительные адреса всех элементов в отдельную область. Это обеспечивает косвенную адресацию и быстрый доступ к каждому значению. Схема такого атрибута представлена на рис. 4.

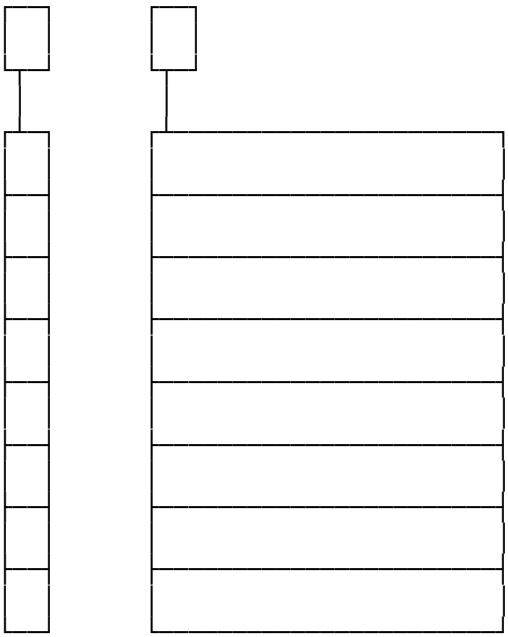


Рис. 4. Схема атрибута произвольного размера

В левой части находится область адресов, а в правой части область значений. Можно отметить, что левая часть есть ничто иное, как атрибут фиксированного размера, который у нас уже был рассмотрен. Но и правая часть, представляет собой атрибут фиксированного размера. Например, если это строковый атрибут, то элементом его будет символ. Таким образом, получается что атрибут произвольного размера образуется из двух атрибутов фиксированного размера. Исходя из этого факта, создадим ещё один слой - слой физического размещения данных. Пусть элементами этого слоя будут пулы (pools), те самые массивы, о которых шла речь ранее. Тогда атрибут будет типичным статическим контейнером пулов.

Взаимодействие между пулами при обработке запросов представляется, в виде схемы атрибута (свойства атрибута, реализуются схемами распределения сообщений по вложенным пулам). Пока эти схемы обработки сообщений достаточно просты, но теперь у нас есть реальная возможность

строить более сложные атрибуты. Например, при хранении строк можно избежать хранения дублей. Достаточно ввести ещё один пул, который будет хранить счётчик повторений. При добавлении строки, которая уже существует, реального добавление не происходит, а просто увеличивается счётчик в пуле счётчиков и добавляется адрес в пуле адресов (адрес указывает на существующую строку с тем же значением).

Ничто не препятствует и тому, чтобы на уровне атрибута ввести дополнительные пулы для ускорения доступа к данным (индексы). Таким образом можно получить целое семейство атрибутов одного типа данных, где каждый представитель этого семейства обладает теми или иными возможностями и особенностями. При этом нам не надо вносить какие-либо изменения в интерфейс между атрибутами и отношениями. Другими словами, разработчики атрибутов могут исследовать различные способы организации атрибутов, не мешая при этом тем, кто реализует уровень отношений. В свою очередь, теперь можно совершенствовать и способы физического хранения информации (пулы), при этом те, кто совершенствуют атрибуты даже не узнают об изменениях способа хранения. Это и есть реализация правила локализации уровней.

9

Прежде чем переходить к рассмотрению более высоких логических уровней, мне бы хотелось вернуться к примеру строкового атрибута, рассмотренного в прошлом письме. В этом примере есть один нюанс, который весьма важен для дальнейшего повествования. Дело в том, что убрав повторяющиеся строки из пула строк, мы, тем самым, допустили, чтобы мощность отдельного атрибута не была равна мощности отношения в целом. Предложенная ранее схема иллюстративна, но для работы не очень удобна. Действительно, после того, как найдена искомая строка нужна определить, какие кортежи на неё ссылаются (в состав каких кортежей она входит). Для этого придётся сканировать пул адресов. Процедура эта хоть и простая, но требует времени. Поэтому имеет смысл ещё немного усложнить атрибут, а именно ввести в него два новых пула. Первый пул будет содержать списки кортежей, в которые входит та или иная строка, а второй пул адреса этих списков. Поддержание адресов списков необходимо, поскольку различные строки могут встречаться различное количество раз, а, следовательно, и списки будут иметь различный размер. Тогда атрибут примет вид, показанный на рис. 5.

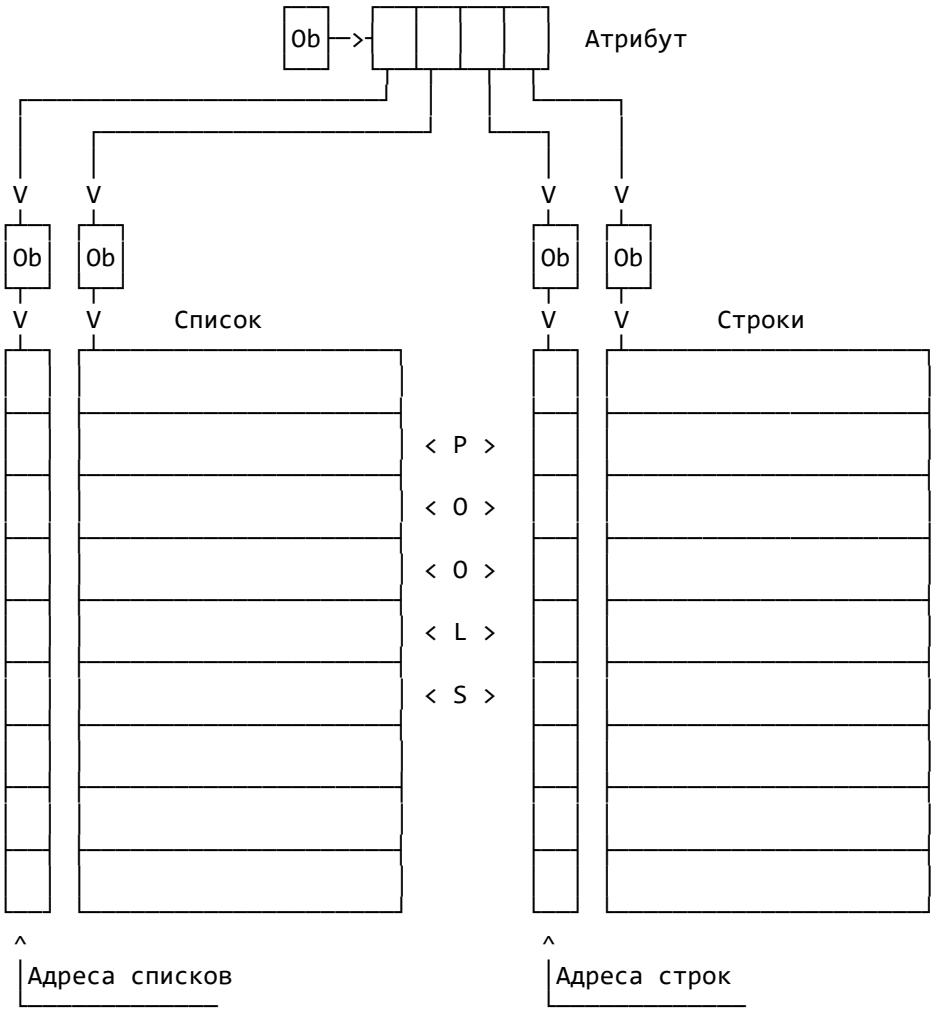


Рис. 5. Вариант строкового пула

Пулы списков (Lists) и их адресов (List's addresses) можно назвать сегментом сопряжения. Благодаря им можно выполнить простой переход от размерности атрибута к размерности отношения. Работает такой атрибут достаточно просто. После того, как найдена нужная строка в пуле строк, извлекается адрес списка из пула списочных адресов. Далее из пула списков получается список кортежей, содержащих данную строку в данном атрибуте. Возможно, такая конструкция атрибута покажется Вам слишком громоздкой, но главное, что хотелось пояснить на данном примере, это понятие сегмента сопряжения, которое понадобится в дальнейшем, а также показать возможности по конструированию атрибутов (в более общем случае, возможности по конструированию любых статических контейнеров).

Более высокие логические уровни

Если Вы обратили внимание на рис.5, то возможно заметили, что объект-атрибут имеет список ссылок на вложенные в него объекты. Все ссылки имеют один и тот же размер, что позволяет представлять объект-атрибут, как пул с элементами постоянного размера. Вполне очевидно, что и любой иной контейнер может иметь аналогичную структуру, к какому бы логическому уровню он не относился. Из чего следует, что названия: "атрибут", "отношение" и "база данных", это синонимы одного и того же вида. Это почти так. Физическое устройство этих объектов действительно одинаковое, но каждый из них имеет свою семантическую окраску, которая

выражается в логике схем. Однако единое физическое устройство позволяет предположить и наличие общих свойств. Так например, следующие SQL операторы на физическом уровне могут обрабатываться одним и тем же кодом:

```
INSERT INTO <table> …
ALTER TABLE <table> ADD <column>
CREATE TABLE <table> ...
```

Во всех этих случаях произойдёт вставка значения в пул на соответствующем логическом уровне (атрибут, отношения, база данных). Занесение данных в пул - это конечное действие, завершающее операцию. Ему, как правило, предшествуют другие действия, различные на каждом логическом уровне. Но эти действия выполняются в рамках схем (свойств контейнеров), и они скорее конструируются, нежели кодируются. При этом какой бы сложной не была логика того или иного уровня, она всегда раскладывается на простые операции - свойства элементарных классов. А элементарными классами в нашем случае являются пулы.

Сейчас можно сделать вывод о том, что реализация элементарных классов позволяет построить любые логические уровни. Единственное, что остаётся реализовать, это схемы (распределение запросов по свойствам вложенных классов) для каждого логического уровня. Но схемы можно и нужно конструировать, а не кодировать. (Более подробно этот вопрос я разбираю в статье по объектному проектированию и здесь мне бы не хотелось на него отвлекаться). Следовательно, кодирование в том виде, котором мы его привыкли видеть, нужно только для реализации свойств пулов и механизма диспетчеризации запросов на вложенные классы. Как отмечалось ранее, пока, на стадии моделирования, пулы реализуются как динамические массивы. Вот и возникает вопрос, язык какого уровня необходим и достаточен для реализации тех немногочисленных свойств пулов, которые перечислялись ранее?.. При переходе от модели к реальному проекту способ хранения претерпит существенные изменения, но эти изменения не слишком отягощают логику и код свойств пулов. С другой стороны, многократное использование кода требует его максимальной эффективности и компактности. То есть, применение низкоуровневых инструментальных средств в данном случае не только возможно, но и желательно.

10

В предыдущем письме был очень кратко рассмотрен вопрос конструирования. Он имеет ключевое значение при проектировании практически любой сложной системы. Дело в том, что благодаря конструированию становится возможным, во-первых, внесение изменений непосредственно в работающую систему (вспомним то, что говорилось про плавный переход от модели к промышленному изделию). Во-вторых, элементы конструктора воплощают сущности предметной области и допустимые действия (интерфейс классов) над ними, что позволяет отразить семантику понятий предметной области на каждом из логических уровней проектирования.

Рассмотренные примеры операторов, показывают полезность введения синонимов для одних и тех же сущностей и действий на различных логических уровнях. Если полиморфизм позволяет единообразно представить различные действия над различными классами, то синонимы помогают различать одно и тоже действие или одни и те же классы, существующие на различных логических уровнях. Следствием этого является значительное сокращение количества элементарных классов и упрощения их сути практически в любой предметной области. Не менее важно и то, что конструирование позволяет вводить высокоуровневые средства разработки, которые зависят только от интерфейса элементарных классов, способа и инструментов агрегации, но не зависят от аппаратной части или операционной системы. В результате можно строить собственные эффективные и портируемые сложные системы.

Однако есть одна важная деталь, которую трудно различить мимолётным взглядом. Действительно можно подменять вложенные в контейнеры классы непосредственно при исполнении системы. Например, так можно решить проблему расширения размера пула целых чисел, если становится недостаточно существующей разрядности. Можно определить схему переноса информации из одного формата в другой и подменить один объект другим. Аналогично решается проблема замены одного атрибута другим, с иным числом пулов и/или с иной логикой работы. Вопрос только в том, кто должен делать замену? Должен ли пул сам уметь менять свой формат, должен ли атрибут сам уметь менять свой тип (класс)? Или это должны уметь делать их контейнера? Ни то, ни другое решение не кажется мне верным. Процесс обслуживания каждого уровня лучше вынести в отдельную область. Эта часть должна отвечать за модернизируемость системы. Дело в том, что при разработке конкретных классов (пулов, атрибутов и пр.) мы не можем предусмотреть все возможные их пути развития. Следовательно, нам не удастся и определить все необходимые средства миграции этих классов из одного представления в другое. То же самое справедливо и для контейнеров этих классов. Кроме того, понятие миграции, как правило, не отражается на семантике класса. Например, двигатель в автомобиле не может заменить себя другой моделью, но это не может сделать и автомобиль. (Я прошу прощения за столь долгое отступление, но мне оно показалось необходимым для дальнейшего изложения. Но пора вернуться к проекту СУБД)

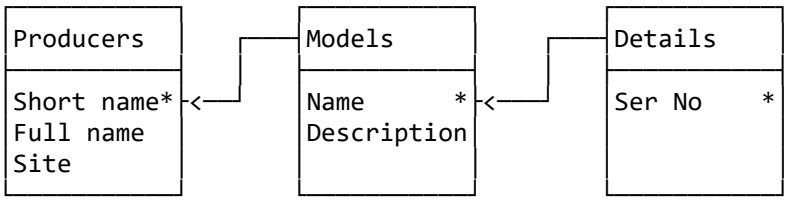
Отношения

Отношения являются контейнерами атрибутов. Однако, если атрибут - это типичный представитель статического контейнера, то отношение - это типичный представитель динамического контейнера. Другими словами, отношение не различает ролей атрибутов, хотя для пользователей и приложений эти роли важны. Отношение предоставляет простой способ группировки атрибутов по семантическим признакам. Соответственно к такой группе атрибутов могут применяться ограничения, права доступа и использования, обработка событий, включая рассылку сообщений при наступлении того или иного события. Возможно, это покажется странным, но отношение несёт по большей части семантическую нагрузку, но не имеет серьёзной нагрузки на уровне реализации. Действительно, схемы обработки сообщений (запросов) для отношения не нужны, поскольку логика обработки запроса заложена в самом запросе. Отношение должно уметь "проигрывать" запросы.

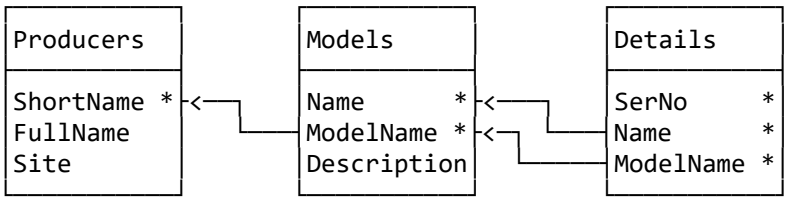
Когда поступает запрос, отношение распределяет его по вложенным атрибутам так, как это указано в самом запросе. Запросы могут быть простыми и комбинированными, объединяющими в себе несколько элементарных запросов, которые были рассмотрены ранее. Запросы могут быть обращены к одному или более отношения или другой сущности базы данных. Многообразие и возможная сложность запросов ставит вопрос о необходимости оптимизации исполнения запросов. Здесь есть две крайности. Первая из них - отсутствие какой-либо оптимизации. Действительно, если допустить возможность параллельного исполнения подзапросов каждым атрибутом, участвующим в запросе, то в этом случае оптимизация практически не нужна. Мало того, она может иметь отрицательный эффект из-за связанных с ней накладных расходов.

Другая крайность - это детальная оптимизация. В этом случае отношение должно хорошо представлять устройство каждого вложенного в него атрибута, его мощность, наличие индекса или иного способа ускоренного доступа. Но детальное знание устройства вложенных атрибутов может нарушить правило локализации, что, в свою очередь, может затруднить модификацию отношения: добавления новых или удаления существующих атрибутов и ограничений. Усложнение логики модификации отношений связано, например, с тем, что эти изменения могут потребовать перенастройки правил оптимизации запросов.

Отношения могут быть связаны между собой посредством механизма уникальных (unique) (первичных (primary)) и внешних (foreign) ключей (key). Любое отношение может иметь произвольное количество связей с произвольным числом отношений. Связь осуществляется посредством дублирования структуры полей уникального (первичного) ключа в структуре внешнего ключа. При каскадных связях количество ключей, образующих ключ, увеличивается, что может привести к очень большому дублированию информации. Для примера можно рассмотреть следующую схему: есть отношение производителей комплектующих изделий для компьютеров, каждый производитель может производить несколько моделей комплектующих, наконец, возможно много изделий данной модели оборудования. Что показано на рис. 6.



Фрагмент инфологической модели



Фрагмент даталогической модели (звёздочкой отмечены поля, входящие в ключ)

Рис. 6. Каскадная схема образования ключа

11

Прежде всего, мне бы хотелось принести извинения за опечатку, допущенную в предыдущем письме ("Проектирование 10"). На рис.6 поле ShortName отношения Producers должно быть связано с полем Name отношения Models, а не с полем ModelName, как изображено на рисунке.

Хранение информации в отношениях не по кортежам, а по атрибутам, наряду с конструированием отношений, даёт интересную возможность рассматривать атрибуты, составляющие ключ, отдельно, вне отношений, по крайней мере, на уровне логики. Ранее уже рассматривалась ситуация, когда мощность отдельного атрибута может быть не равна мощности отношения, содержащего данный атрибут. Здесь ситуация аналогична. С точки зрения отношения, имеющего группу атрибутов ключа, в качестве первичного ключа, мощность ключа равна мощности отношения, то есть, количество кортежей в отношении равно количеству кортежей ключа. Иначе обстоит дело с теми отношениями, которые содержат ту же группу атрибутов, в качестве внешнего ключа. При наличии связи один-ко-многим мощность такого отношения может быть больше, чем мощность ключа. Если вернуться к рассмотрению примера, изложенного в предыдущем письме, то можно сказать, что моделей изделий, выпускаемых фирмами-производителями, значительно больше, чем самих фирм-производителей. Как следствие, в отношении "Models" названия фирм-производителей (внешний ключ) будут дублироваться. Например, одна и та же фирма-производитель мониторов может выпускать целую линейку моделей мониторов различных размеров и качественных характеристик.

Если рассматривать ключ, как самостоятельный элемент (на логическом уровне), то в этом случае необходимо выполнить сопряжение между мощностью ключа и мощностью тех отношений, где этот ключ является внешним. Воспользуемся уже известным решением -

сегментом согласования. В этом случае между ключом и отношением создаётся список, связывающий каждый кортеж ключа с теми кортежами отношения, которые содержат данный кортеж ключа.

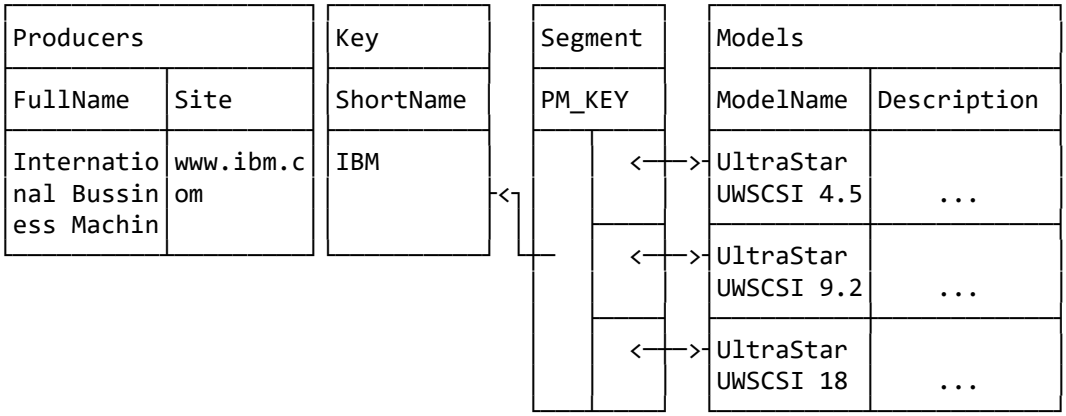


Рис.7. Реализация связи между отношениями

Связь между отношением и внешним ключом является двунаправленной связью. Для простоты можно считать, что вместо внешнего ключа отношение Models содержит ссылку на соответствующий кортеж отношения Producers. Реализация механизма ссылочной целостности является внутренним механизмом СУБД и этот механизм недоступен извне. Разработчики БД не должны иметь доступа к этому механизму и прямое установление ссылок (непосредственное манипулирование значениями полей ссылок) является недопустимым. Работа с ключами на внешнем уровне происходит в соответствии с требованиями реляционной модели.

Аналогично тому, как реализована связь между отношениями Producers и Models можно достроить связь между отношениями Models и Details. Поле ModelName входит в состав первичного ключа, а, следовательно, может участвовать в образовании связей с другими отношениями, как это и происходит в случае введения отношения Details. Построение связи происходит полностью аналогично тому, как это представлено на рис. 7. Сегмент согласования теперь принимает ярко выраженную каскадную (иерархическую) форму, как это видно из рис. 8.

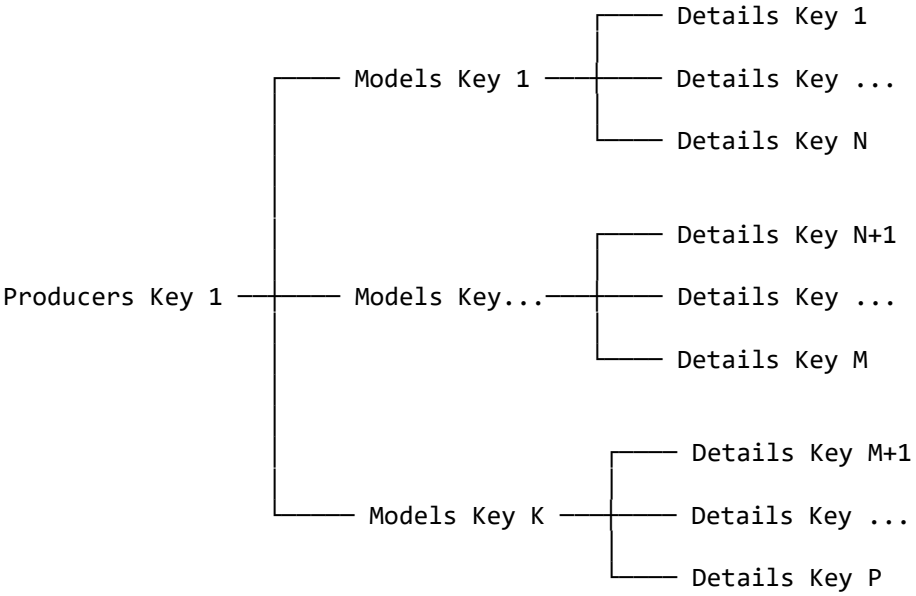


Рис. 8. Образование каскадной схемы ключа

Наверное будет не лишним ещё раз отметить, что каскад образуется на сегменте согласования,

то есть на ссылках, но не на значениях атрибутов. Сами атрибуты по-прежнему принадлежат отношениям. Каждое отношение может иметь несколько точек входа в сегмент согласования, так отношение Models может входить в сегмент согласования через Models Key или через Producers Key, а отношение Details может входить также через Details Key. Это фактически аналогично тому, что отношение Models имеет в своём составе поле Name (references Producers (ShortName)), а отношение Details имеет поля ModelName и Name (references Model (ModelName, Name)).

Иерархическая схема сегмента согласования является частным случаем сетевой схемы. Если мы введём в рассмотрение дополнительные отношения, имеющие связь с тремя данными отношениями, то не исключено, что иерархия, в чистом виде, будет утрачена. Это тем более справедливо, если учесть, что не всегда внешний ключ входит в состав первичного ключа. Логика связей задаётся предметной областью и формируется в процессе нормализации. В общем случае внешний ключ может быть:

1. Необязательным (nullable);
2. Обязательным (not null);
3. Входящим в состав первичного ключа.

В соответствии с этим может строится работа по поддержанию ссылочной целостности, как это определено в стандарте SQL 92. При удалении первичного ключа в случаях 2 и 3 должно произойти удаление и связанных с ним кортежей подчинённых отношений и соответствующих списков ссылок из сегмента согласования. Для случая 1 допустимо установить значение внешнего ключа в положение null, что на физическом уровне означает обнуление ссылок. При изменении значения первичного ключа внешние ключи могут быть каскадно обновлены, что не потребует изменения ссылок. Внешние ключи с соответствующими им кортежами могут быть удалены, что повлечёт удаления как кортежей из подчинённых отношений, так и списков связей из сегмента согласования. Наконец, внешние ключи, в третьем случае, могут быть обнулены, что потребует обнуления ссылок.

Предложенная схема интересна ещё и тем, что она соответствует не только реляционной модели, но и сетевой. Фактически разница только терминологическая. Если заменить понятие "отношение" понятием "набор записей", то данная модель хранения информации достаточно полно соответствует сетевой модели CODASYL. Наконец, из того материала, что нам ещё предстоит обсудить, будет видно, что и, так называемая, объектная модель реализуется в этой модели хранения информации достаточно просто и эффективно. Но, прежде, чем переходить к рассмотрению этих вопросов необходимо рассмотреть вопросы, связанные с ускорением доступа к кортежам, то есть вопросы индексации.

12

Индексация представляет собой один из наиболее популярных способов ускорения доступа к данным. Существует достаточно много способов индексации, большая часть которых основана на построении древовидных структур. В настоящее время приобрели популярность, так называемые, битовые (двоичные) индексы. И, по прежнему, хотя и относительно редко, для ускорения доступа к данным используют хэш-функции.

Наиболее популярные древовидные структуры индексов, в свою очередь, можно разделить на две большие группы: индексы, которые дублируют данные и индексы, которые ссылаются на данные. На первый взгляд, индексы дублирующие данные должны быть более производительны, чем индексы, которые ссылаются на данные. Повышение производительности связано с уменьшением обращения к данным. Пока данные хранятся по кортежам, это утверждение _почти_ справедливо. Действительно, ведь для того, чтобы обратиться к кортежу его скорее всего

потребуется подгрузить с диска или скопировать из кэш-буфера. Такая подгрузка может существенно замедлить процесс поиска. Однако при допущении, что данные уже находятся в памяти, ситуация может быть принципиально иной.

В свою очередь, размещение данных не по кортежам (строкам), а по атрибутам (столбцам) даёт ряд дополнительных преимуществ. Во-первых, даже если данных нет в памяти, их подгрузка существенно проще, поскольку подгружается не весь кортеж, а только те атрибуты, которые участвуют в процессе поиска. Во-вторых, как отмечалось ранее, индексированные атрибуты могут иметь только уникальные значения даже для индексов, не объявленных уникальными (UNIQUE). В-третьих, данные в атрибуте могут хранится не в естественном, а в нужном (отсортированом) порядке, что также минимизирует потребность в подгрузке страниц из кэша или с диска. Наконец, в-четвёртых, при построении индекса, более чем по одному атрибуту, возможно проведение параллельного поиска в каждом из атрибутов, образующих индекс.

Как следствие данного подхода можно отметить, что это решение обеспечивает высокую компактность хранения, отсутствие дублирования данных не только в индексе, но и самом атрибуте (хранение только уникальных значений). Компактность хранения и использование древовидных индексов небольшой ширины, но зато большой глубины, а также селективная подгрузка данных, дают ощутимое увеличение производительности. Эти рассуждения полностью применимы и для двоичной индексации.

Объединение в индексе, более чем одного атрибута, требует дополнительной структуры данных. В рамках этой структуры устанавливается соответствие между полями атрибутов, принадлежащих одному кортежу. Этот шаг необходим, поскольку данные разнесены по различным атрибутам, да ещё и сохранены произвольным образом. Если с логического уровня спустится на физический, то есть на уровень реализации, то можно отметить, что данная структура производит выравнивание мощностей атрибутов, входящих в индекс. Если индекс объявлен уникальным, то его мощность будет совпадать с мощностью отношения, иначе индекс будет содержать либо дублирующиеся кортежи, либо сегмент согласования. В этом письме я не привожу рисунки и схемы, поскольку всё, что ранее говорилось об отношении полностью справедливо и для индекса. Просто индекс играет роль отношения для вложенных в него атрибутов. Это важно отметить, поскольку даёт представление о том, как часто на высоких логических уровнях используются одинаковые механизмы, даже не смотря на то, что нижние логические уровни различаются весьма существенно. Именно этому вопросу будет посвящено следующее письмо.

13

На данный момент мы прошли достаточно большой путь, определив несколько логических уровней. Определение и спецификация логических уровней - это один из наиболее ответственных этапов проектирования. Вполне возможно, что у Вас остались вопросы, но их можно разрешить на самой примитивной модели, о которой я говорил в самом начале повествования. Напомню, что каждый логический уровень независим от других логических уровней и взаимодействует с ними исключительно на основе специфицированного интерфейса. Независимость логических уровней даёт возможность развивать каждый уровень, сообразуясь только с логикой данного уровня. При проектировании сложной системы - это одно из важнейших положений.

Прежде, чем переходить к дальнейшему обсуждению, следует ещё раз остановиться, чтобы определить положение каждого уровня. Рис. 9 представляет основные логические уровни, расположенные ниже уровня отношений. Если по какой-то причине данных уровней окажется недостаточно, то можно, не нарушая логики существующих уровней, внедрить недостающие уровни. Как это происходит? Об этом и пойдёт речь ниже.

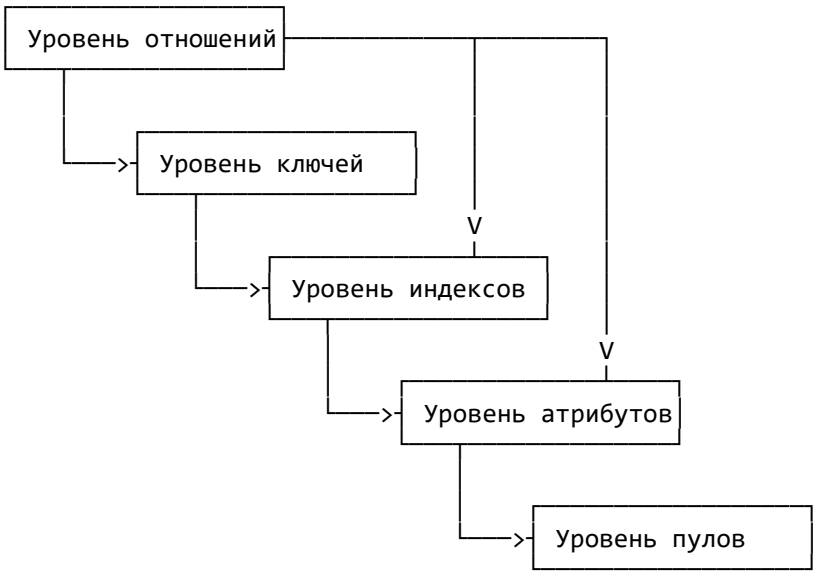


Рис. 9. Логические уровни расположенные ниже отношения

Следующим, за отношениями, логическим уровнем должен быть уровень базы данных. Так реализовано большинство коммерческих СУБД. Но мы не будем спешить. В практических задачах очень часто встречается ситуация, когда желательно классифицировать информацию. Например, есть сущность "товар", которая объединяет общие атрибуты любого товара (вид товара, поставщик, цена и дата поставки и т.п.), но частные виды товаров имеют свои атрибуты, которые могут быть важны. Например, такой товар, как одежда и обувь характеризуются размером, и этот атрибут должен быть обязательным. Для продуктовых товаров важен предельный срок реализации, а для молочных продуктов, ещё и жирность, для алкоголя - содержание спирта, для мебели вид отделочного материала и т.д. Как организовать хранение такой информации? Можно все частные атрибуты свести в одно отношение. Но такое решение нельзя признать хорошим. Действительно, если жирность важна для молочных продуктов, то этот атрибут теряет смысл в приложении к одежде, обуви, мебели и алкоголю. Если мы объявим этот атрибут, как nullable, то нет гарантии, что у партии творога будет указана жирность. Если объявить данный атрибут, как not null, то мы заставим пользователей заполнять этот атрибут для товаров, у которых он не определён. Следовательно, при запросе выдать товары с максимальным значением жирности, можно получить ответ - мебельный гарнитур "Берёзка". (Более подробно эта проблема рассмотрена в моей статье "Объектное представление о реляционной модели").

Рассмотрим решение, доступное на уровне проектирования СУБД. Допустим, что у нас есть некоторый объект, который способен объединять несколько отношений в некое единое представление. При обращении к этому нечто, оно будет перенаправлять запрос к реальным отношениям или сущностям, подобным себе. Назовём это нечто - классификатором. В приведённом выше примере у нас будет классификатор товаров. Совсем не обязательно, чтобы данный классификатор совпадал с общепринятыми классификаторами товаров (это всего лишь не очень удачное название). Классификатор является типичным динамическим контейнером, то есть вложенные в него отношения не обладают какими-то особыми ролями. С точки зрения классификатора, они все одинаковы. Для пользователей классификатор представляется обычным отношением. Например, если мы просим выбрать товары поставляемые определённым поставщиком в заданном интервале времени, то классификатор разошлёт запросы на выборку всем вложенным в него отношениям. В результате пользователь может увидеть список товаров, которые реально принадлежат к различным товарным группам.

Подобные механизмы есть во всех объектно-ориентированных СУБД. При этом классификатор представляется как класс, а конкретные кортежи - объектами. На самом деле, это не совсем соответствует действительности, поскольку здесь присутствует только наследование реализации. Это полезно, но этого маловато для того, чтобы говорить о поддержке объектной технологии. Безусловно, какое-то пользовательское приложение имеет право рассматривать любую информацию в базе данных, как объекты, но для СУБД - это просто информация и не более. Ни о какой инкапсуляции, с точки зрения СУБД, говорить не приходится, поскольку она не только знает структуру классификатора, но и может читать и писать в поля, что недопустимо. Прятать же поля за интерфейсные свойства ещё более бессмысленно, так как накладные расходы на работу с такими данными возрастут многократно. А суть останется той же. Надеюсь, что мы ещё вернёмся к обсуждению объектных СУБД, а пока продолжим рассмотрение устройства классификаторов.

Итак, классификатор описывает общую структуру полей, которые затем реализуются в отношениях или классификаторах-наследниках. Кроме структуры полей, возможно определять на уровне классификатора любые ограничения, связи с другими классификаторами или отношениями, индексы, вычисляемые поля, триггеры и т.д. Все эти составляющие проецируются на отношения или наследуются другими классификаторами. Например, объявление одного поля или комбинации нескольких полей уникальными, мы тем самым требуем, чтобы эта уникальность распространялась на все отношения, относящиеся к данному классификатору. При этом ни одно из отношений не может иметь комбинацию значений этих полей, которая уже существует в другом отношении. Если Вы определили одно из полей инкрементальным на уровне классификатора, то оно будет последовательно выдавать номера независимо от того, к какому отношению они относятся.

При запросе на выборку к классификатору, он передаёт этот запрос вложенным отношениям и классификаторам-наследникам. Выбираются только те поля, которые определены у классификатора, но не все поля, которые определены у классификаторов-наследников. Фактически классификатор, в данном случае, работает как VIEW, основанном на UNION. Но в отличие от VIEW классификатор позволяет прозрачно изменять число вложенных отношений и классификаторов-наследников. После добавления нового отношения или классификатора-наследника они автоматически включаются в список диспетчеризации для данного классификатора.

В следующем письме приводится рис. 10, который показывает устройство классификатора.

14

Графически классификатор можно отобразить так:

Рис. 10. Схема классификатора

Из схемы приведённой на рис.10 видно, что классификатор может содержать либо другой классификатор, наследующий реализацию, либо отношение, имеющее структуру описанную в классификаторе. Рассмотренный выше пример с товарами можно представить следующей схемой:

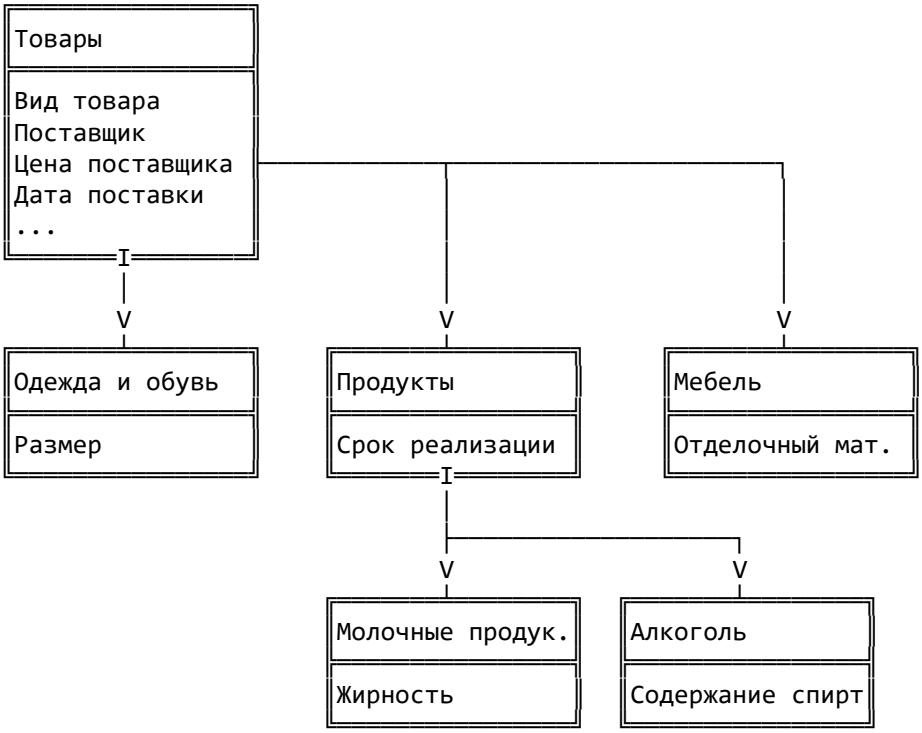


Рис. 11. Схема наследования реализации у классификаторов

Каждый классификатор может иметь или не иметь ассоциированное с ним отношение. Если с классификатором не связано отношение, то такой классификатор можно назвать абстрактным. Для абстрактного классификатора нельзя выполнить операцию вставки, остальные операторы DML, работающие с отношениями, могут применяться и для классификаторов. Как отмечалось ранее, операторы будут транслироваться на вложенные классификаторы и связанные с ними отношения, точно также, как перетранслируется оператор SELECT.

Теперь у нас начал формироваться ещё один логический уровень, расположенный между отношениями и базой данных. Являются ли классификаторы единственными "жителями" данного уровня? Не обязательно. На этом уровне можно расположить и другие сущности оперирующие отношениями, как вложенными классами. Такой сущностью может быть, например, переключатель. Суть переключателя проста. Допустим мы создаём табличную структуру для хранения результатов социологического опроса. Априори известно, что результаты опроса будут обрабатываться раздельно для мужского и женского населения. Обычный способ заключается в создании таблицы, имеющей поле "Пол". Безусловно такая таблица будет работать. Но мы разделим эту таблицу на две части и вложим обе полученные таблицы в переключатель. Пул переключения будет иметь два значения: "М" и "Ж". Если в запросе на вставку указано "Пол" = "М", то данные попадут в одну таблицу, а если поле "Пол" = "Ж", то данные будут помещены в другую таблицу. Значение переключателя всегда имеет атрибут NOT NULL. Само поле "Пол" не включается ни в одну таблицу, а принадлежит переключателю.

При выборке будет иная ситуация, запросы к обеим таблицам будут объединяться по UNION и каждый запрос будет содержать соответствующую константу. То есть, исходный запрос на выборку:

```
SELECT * FROM SR_RESULT;
```

будет преобразован к виду (запись упрощена):

```
SELECT *, 'M' FROM SR_RESULT_M
UNION
```

```
SELECT *, 'Ж' FROM SR_RESULT_F;
```

Возможно пример с данными социологического опроса кажется несколько надуманным, но он иллюстративен. А ситуация, когда одно отношение желательно разбить на несколько (поскольку они редко используются совместно) встречается очень часто. Наверное большинство разработчиков систем сталкивалось с проблемой архивации данных. Здесь под архивацией понимается перенос информации из оперативных таблиц в аналогичные по своей структуре архивные таблицы. С помощью переключателей эта проблема решается достаточно просто. Значения пула переключателя будут содержать даты (или диапазоны дат). Работа с таким переключателем прозрачна для приложений, но она позволяет разводить оперативные и архивные данные, ускоряя тем самым работу с оперативными данными. Другое решение по организации подобных переключателей - это поддержание "плавающих" значений, в частности, тех же дат. В этом случае, возможен автоматический перенос "устаревших" данных. Наконец, есть и другие области, где переключатели могут быть очень полезны. Например, при автоматизации управления предприятием бывает удобно разнести одну и ту же таблицу по структурным подразделениям, но при этом сохранить простую возможность получения общей выборки. Такими таблицами могут быть таблицы служащих, оборудования, материалов и т.п.

Но и переключатели не исчерпывают всех возможностей данного уровня. Не менее полезными в практической работе могут быть и репликаторы. Задачей этих сущностей является синхронизация информации в различных базах данных. При этом информация в каждой базе данных может иметь уникальную структуру. При наступлении заданных условий репликатор соберёт информацию из одной базы данных и разошлёт её по другим базам данным так, как указано в его схеме. Ну, и конечно репликация может быть на основе двухфазного commit, синхронной и асинхронной.

Наконец, на этом логическом уровне обитают всем известные сущности, такие как VIEW и SELECTED STORED PROCEDURE. Если всех перечисленных сущностей Вам недостаточно, то можно создать свои собственные. Ваши возможности ограничены только Вашей фантазией. В самом начале я писал о том, что любая объектная система никогда не находится в завершённом состоянии. Она имеет возможности для развития и эти возможности развития проецируются на все логические уровни, каждый из которых может развиваться совершенно независимо от других уровней. Единственное ограничение - это поддержание неизменности межуровневого интерфейса.

На этом мне бы хотелось завершить рассмотрение данного логического уровня и перейти к рассмотрению логического уровня баз данных. Этому будет посвящено следующее письмо.

15

Базы данных (БД) являются ещё одним примером динамического контейнера. Они включают в себя объекты всех логических уровней, начиная с уровня отношений. Обслуживание всех этих объектов происходит одинаково, на основе общих полиморфных свойств. Под обслуживанием понимается, во-первых, ведение реестра вложенных объектов, что является общей функцией любого контейнера. Во-вторых, проверка прав доступа к тому или иному объекту. И, наконец, в-третьих, сохранение, восстановление и инициализация вложенных объектов.

О способе проверки прав доступа уже кратко говорилось при рассмотрении свойств атрибутов. Как известно, права доступа назначаются с помощью оператора GRANT и отменяются с помощью оператора REVOKE. База данных имеет служебные отношения, где отмечаются права пользователей и/или ролей, а также объектов, по отношению к которым данные права определены. Примерная схема взаимосвязи показана на рис 12.

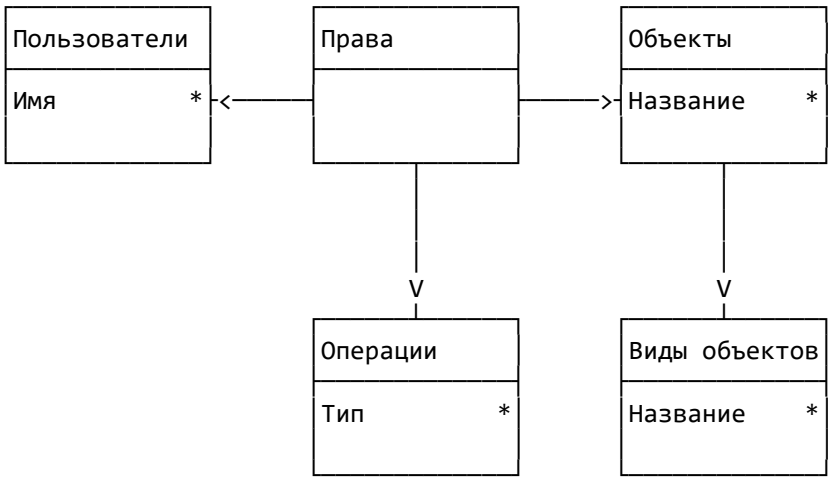


Рис.12. Взаимосвязи между пользователями, объектами БД и операциями

Как видно из рис.12 "Права" образуют тринарную связь между "Пользователями", "Объектами" и "Операциями" (SELECT, INSERT, DELETE и т.д.). Реально каждой операции можно поставить в соответствие некоторую "маску". При получении запроса "маска" операции, указанной в запросе накладывается на маску прав пользователя по отношению к объектам, указанным в запросе. При положительном результате запрос направляется на выполнение, при отрицательном - отклоняется.

Операторы GRANT и REVOKE на физическом уровне реализуются посредством операций вставки и обновления системного отношения "Права" пользователя на доступ к объектам БД. То есть, эти операторы не требуют каких либо дополнительных свойств, кроме тех, что были описаны ранее.

БД отвечает за сохранение, восстановление и инициализацию объектов, находящихся на более низких логических уровнях. Для этого она должна располагать специальными службами, задачей которых является распределение адресного пространства как в оперативной памяти, включая кэши, так и на внешних носителях; поддержание классов объектов; отслеживание использования тех или иных объектов и т.п. Эти службы, в свою очередь сохраняют свою информацию в БД.

В письме "Проектирование 7" на рис. 1 была представлена иерархия атрибутов, в которой, в частности, было предусмотрено место для хранения объектов. Как видно из рисунка "объект" представляет собой атрибут, не имеющий фиксированного размера поля. Это нечто напоминающее BLOb. Ни БД, ни другие, рассмотренные ранее, сущности не знают "устройство" конкретного класса или объекта. То есть, поддерживается полная инкапсуляция (защищенность), как данных, так и кода объектов. При создании нового объекта некоторого класса, он инициализируется по шаблону, описанному в классе (default values). Существующие постоянные (persistent) объекты сохраняются в БД и восстанавливаются из неё. И при создании и при восстановлении объектов из БД владельцу передаётся физический или логический адрес. Логические адреса используются для организации шлюзов при работе с объектами расположенными в других процессах (разделяемые объекты), в том числе и для работы с удалёнными (remove) объектами, находящимися на других компьютерах.

Следует отметить, что представление объекта в виде одного простого атрибута, это не более, чем упрощение, которое было введено, дабы не перегружать текст излишними деталями, пока в них не возникнет потребность. Конечно, объект состоит из кода и данных и, если данные являются приватной частью каждого объекта, то код разделяется объектами одного или более классов. Это должно быть учтено при разработке.

Сейчас наверное имеет смысл рассмотреть как происходит обработка запроса, поступившего к

БД. После того, как сверены объекты запроса и проверены права доступа к ним, строится схема исполнения запроса. Схема выполнения запроса - это упорядоченное и связанное множество подзапросов к сущностям уровня отношений, классификаторов, репликаторов, переключателей, view и хранимых процедур. Упорядочение множества подзапросов не означает их последовательного выполнения. Здесь полностью применимо все то, что говорилось о схемах контейнеров. Строго говоря, запрос представляет собой _временный_ статический контейнер. Современные оптимизаторы запросов используют весьма сложные механизмы с целью ускорения времени исполнения запроса. Это оправдано, поскольку сами данные хранятся далеко не оптимально. Если же рассматривать схему параллельного исполнения подзапросов, причём параллельность простирается вплоть до атрибутов, которые задействованы в подзапросе. Плюс к этому использовать хранение только уникальных значений атрибутов, то попытка оптимизации может только напрасно отнять время. Однако параллельная схема обработки запросов будет эффективной только на системах с массовым параллелизмом.

Наверное будет не лишним отметить, что схема обработки запроса представляет собой объект, расположенный на более низком логическом уровне, чем БД, но выше уровня отношений, классификаторов и пр., так как показано на рис.13.

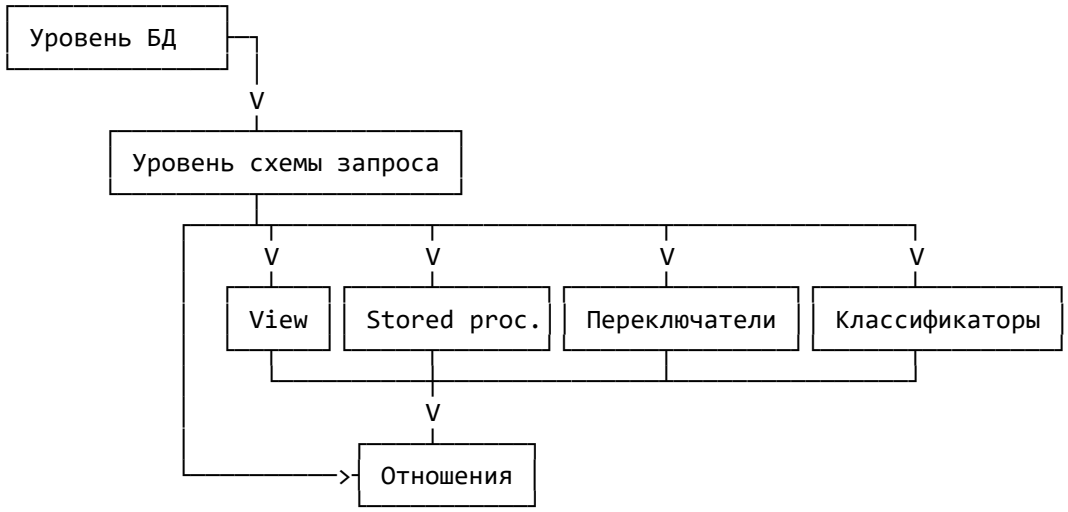


Рис. 13. Логические уровни выше уровня отношений