**Российская академия наук**
**Сибирское отделение**
**Институт систем информатики**
**им. А. П. Ершова**

**И. В. Кречетов**

# АВТОМАТИЧЕСКОЕ ПОСТРОЕНИЕ SQL ЗАПРОСОВ ДЛЯ ОБЪЕКТНЫХ ДАННЫХ

**Препринт**
**118**

**Новосибирск 2004**

Задача генерации запросов возникает при реализации подсистемы хранения для объектно-ориентированного приложения, использующего реляционную СУБД. Предлагается набор успешно опробованных на практике проектных решений, затрагивающих следующие аспекты: гибкость установки соответствий между объектами и таблицами, минимизация количества обращений к базе данных, независимость от диалекта SQL, параметризация и кэширование запросов.

© Институт систем информатики им А. П. Ершова СО РАН, 2004

**Siberian Division of the Russian Academy of Sciences**
**A. P. Ershov Institute of Informatics Systems**

**I. V. Krechetov**

# THE AUTOMATED BUILDING OF SQL QUERIES FOR AN OBJECT DATA

**Preprint**
**109**

**Novosibirsk 2004**

The problem of query generation arises in the context of a persistent layer implementation for an object-oriented application which utilizes a relational DBMS. Here we propose a set of software design solutions successfully approved in practice. These solutions cover the following aspects: flexibility of the object-relational mapping, minimization of the quantity of database calls, independence from a SQL dialect, query parameterization and caching.

© A. P. Ershov Institute of Informatics Systems, 2004

# 1. ПОСТАНОВКА ЗАДАЧИ

Мы рассмотрим некоторое объектно-ориентированное приложение, реализованное на универсальном языке программирования (таком, как С++, Java или С#), использующее реляционную базу данных в качестве основного хранилища данных и ориентированное на обработку транзакций. В качестве интерфейса взаимодействия с реляционной СУБД будет использоваться структурированный язык запросов SQL.

Мы ставим перед собой задачу построения программного средства автоматической генерации SQL запросов, манипулирующих объектными данными приложения. Искомый программный инструмент должен удовлетворять следующим требованиям.

- *Операционная полнота*. Поддержка всех возможных операций над данными объектов: вставка, обновление, выборка, удаление.
- *Универсальность*. Не допускается привязка к некоторому конкретному диалекту языка SQL.
- *Минимизация количества запросов для передачи объектной информации*.
- *Гибкость*. Возможность использования специфических сервисных функций, предоставляемых некоторыми СУБД.
- *Удобство сопряжения*. В область нашего внимания попадает не только сам генератор запросов, но и смежные с ним компоненты приложения (итераторы, proxy-объекты и пр.).

Предлагаемое проектное решение будет проиллюстрировано элементами объектной модели реализованного автором обобщённого уровня хранения [1].

# 2. КОНТЕКСТ

Ниже мы подробно опишем, в каком контексте работает автоматический построитель SQL-запросов.

## 2.1. Отображение

Если построение системы соответствует объектной парадигме программирования, то хранимые данные такой системы являются объектами. При этом, носителем информации, обеспечивающим её сохранность между за-

пусками приложения или пользовательскими сессиями, является инструмент, следующий другой парадигме — реляционной. Возникает необходимость сопряжения двух различных моделей данных; требуется совершать отображение (*mapping*) структуры объектов в схему реляционной базы данных[2].

Задача отображения имеет два аспекта: статический и динамический. Статическое отображение подразумевает сопоставление определённой иерархии классов с их атрибутами и ассоциациями набору реляционных таблиц, связанных внешними ключами, в то время как динамическое отображение сопоставляет экземпляры классов и конкретные строки таблиц реляционной базы данных.

Динамическое отображение индуцировано статическим и призвано во время выполнения программы сохранять такие свойства объектов, как идентичность (*identity*) и полиморфизм [3].

### 2.2. Метамодель данных

Запросы к реляционной СУБД необходимы для операций с данными объектов. Требуется загружать, сохранять, создавать новые данные, делать выборки по определённым критериям, удалять объекты. Все эти действия в конечном итоге сводятся к SQL запросам данных реляционных таблиц. Во время построения запроса нужно «знать всё» об отображаемых в строки таблиц объектах: идентификатор, класс, имена и типы атрибутов, средства для чтения и записи атрибутов, связи с другими объектами, их множественность, средства для чтения и установки связей. Всю эту информацию предоставляет метамодель объектных данных.

Метамодель — ключевое звено в процессе автоматизации хранения данных объектно-ориентированного приложения. Она представляет собой набор классов для описания объектной модели приложения, таких как: «Предметная область», «Класс», «Атрибут», «Ассоциация», «Идентичность». Любой хранимый объект предоставляет своё мета-описание для построения запросов к СУБД.

Обычно устанавливаются некоторые правила отображения метамодели в структуру реляционных таблиц. Используются фиксированные правила именования элементов модели, сопоставление *конкретных типов данных* приложения и типов данных СУБД.

Многие универсальные языки программирования обеспечивают средства для работы с метамоделью. Если же такие средства отсутствуют, то про-

граммист, разрабатывающий уровень хранения, самостоятельно реализует требуемые абстракции. Именно на этом случае мы и остановимся.

### 2.3. Обобщенное хранилище

Мы будем рассматривать автоматический генератор SQL запросов в контексте обобщенного хранилища данных объектно-ориентированного приложения [1], когда реляционная база данных является основным, но далеко не единственным носителем информации. Присутствуют и другие механизмы хранения и представления объектных данных, а именно: файловая система, система контроля версий, удобное для чтения человеком текстовое представление. Кроме различных по типу механизмов хранения и представления информации возможно одновременное присутствие нескольких версий механизма определённого типа, например, несколько реляционных баз данных, которые могут использовать разные диалекты языка запросов SQL.

### 2.4. Порядок взаимодействия с БД

От механизма хранения данных вообще и от реляционной БД в частности мы будем требовать лишь поддержку состояния экземпляров классов-сущностей. Поведение объектов (их методы или операции) реализуется на универсальном языке программирования, и этот код исполняется на некоторой изолированной рабочей станции или сервере приложений, обслуживающем клиентские запросы.

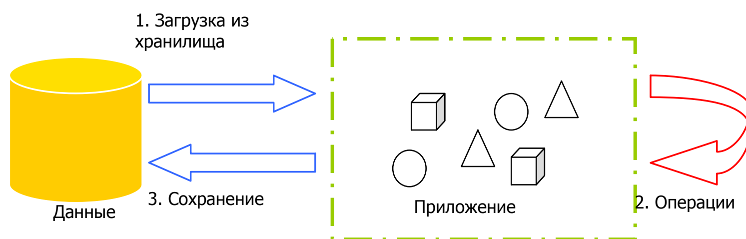


**Рис. 2.1.** Использование сущностей

Как показано на рис. 2.1, операции выполняются в области памяти приложения (назовём эту область *доменом*) и должны быть реализованы на

универсальном объектно-ориентированном языке программирования. Помимо операций в домене должно присутствовать полное описание «сущностных» классов со всеми атрибутами и ассоциациями — статическая метамодель, или *словарь* данных, в то время как описанием для поведения будет код методов на Java или C++, необходимый для исполнения бизнес-приложения.

# 3. РЕАЛИЗАЦИЯ

## 3.1. Сценарии

Перед описанием проектного решения и реализации мы выделим типовые сценарии, в которых используется генератор запросов для манипулирования данными приложения. Все перечисленные действия должны быть отражены в постоянном хранилище данных.

- Сохранение нового объекта.
- Загрузка объекта определённого типа, имеющего данный идентификатор.
- Обновление значений атрибутов объекта.
- Изменение связи: добавление или удаление связанных объектов. Каждая связь объекта является экземпляром межклассовой ассоциации.
- Выбор множества объектов по данному критерию.
- Удаление объекта.

## 3.2. Распределение ролей

Для объектно-ориентированной реализации предлагается набор классов, обеспечивающий работу нескольких программных уровней. Таким образом, группы классов составляют пакеты или модули по признаку *функционального зацепления*.

Уровень бизнес-логики основывается на взаимодействии хранимых сущностей (объектов предметной области) и некоторых синтетических объектов в рамках сценариев, реализующих поведение программной системы. При этом сохранность экземпляров сущностей осуществляется посредством отправки им простых сообщений *сохранить*, *загрузить*, *удалить*. И все хранимые объекты обращаются к единственному в системе брокеру хранения для выполнения запрошенных операций [4].

Брокер хранения выступает в роли посредника между объектами и механизмами хранения, которые для них используются. Все операции записи и чтения состояния объектов делегируются единственному брокеру без знания того, какой механизм используется в данный момент.

Нас будет интересовать случай, когда механизмом хранения является реляционная база данных. Для иллюстрации взаимодействия объектов рассмотрим процесс сохранения сущности, не имеющей связей с другими объектами. Таким образом, состояние данного экземпляра определяется только значением атрибутов, которые и будут записаны в базу данных.

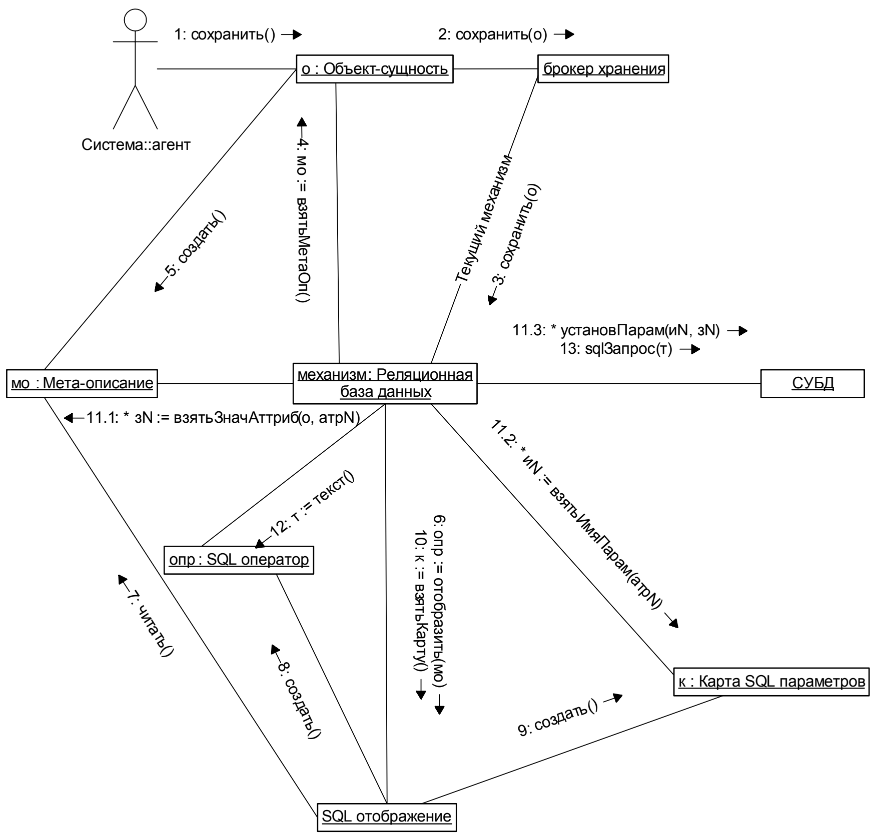


**Рис. 3.1.** Взаимодействие при сохранении объекта

Рис. 3.1 является UML-диаграммой взаимодействия [5] и передаёт некоторые ключевые моменты сохранения состояния объекта в реляционной базе данных. Аналогичное сотрудничество объектов имеет место при наличии связанных сущностей для сохраняемого экземпляра. Ниже мы рассмотрим архитектуру классов для использованных в диаграмме объектов, а также, более подробно, генерацию SQL-отображений по мета-информации и обработку связанных объектов.

## 3.3. Мета-описание

### *3.3.1. Домен*

Объект «Домен» содержит ссылки на все классы сущностей системы, т.е. на экземпляры типа «Хранимый класс», являющиеся мета-классами для сущностей приложения. Перед началом работы приложения, использующего объектное обобщённое хранилище данных, необходимо создать *домен* и заполнить его описанием всех классов.

Как и *брокер хранения*, *домен* существует в единственном экземпляре во время выполнения программы. Потребность в данном объекте возникает при отсутствии в используемом языке программирования средств получения полной информации о типе во время выполнения (*RTTI — Run-time Type Information*). В этом случае требуется некоторый контейнер для всех мета-классов предметной области, обращение к которому происходит для получения конкретного мета-класса, имеющего указанное имя. При этом, каждый хранимый тип объектов полиморфно предоставляет имя своего класса в *домене*.

### *3.3.2. Класс*

Объект типа «Класс» является классом хранимых объектов, а с точки зрения некоторого объекта системы, подлежащего хранению, это его мета-класс.

Задача *класса* — предоставлять информацию об атрибутах и ассоциациях сущностей предметной области, в частности, об отношении наследования к другим *классам*. У *класса* есть имя, и он способен по требованию создавать новые экземпляры соответствующих ему сущностей.

### *3.3.3. Атрибут*

Атрибут представляет собой поле данных сущности. Атрибуты хранимых объектов имеют в приложении *конкретный* тип данных, который взаимно-однозначно соответствует определённому типу данных табличной колонки в СУБД.

### *3.3.4. Однонаправленная ассоциация*

Межклассовая ассоциация указывает на наличие важной с точки зрения приложения семантической связи между сущностями. Связь между двумя конкретными объектами всегда является элементом соответствующей ассоциации их классов.

Для простоты мы будем рассматривать лишь однонаправленные бинарные ассоциации. Только такие отношения будут подлежать хранению. Более того, мы ограничиваемся лишь следующими значениями кардинальности:

- один к одному,
- один к нулю или одному,
- один ко многим.

Специально выделяются случаи, когда ассоциация является агрегацией или композицией. Ещё более специальным является вариант отношения «*is a*», т.е. наследование классов.

### *3.3.5. Настраиваемые атрибуты и связи*

Хранить объекты в реляционной базе данных невозможно без определённых правил отображения. По этим правилам отображения и именования классам сопоставляются таблицы, атрибутам — колонки, ассоциациям — соответствующие отношения таблиц с внешним ключом.

Однако бывают случаи, когда для атрибута или ассоциации невозможно воспользоваться общей схемой отображения. Например, когда значением атрибута является большой текст или графическое изображение и реляционная СУБД не позволяет возвращать такие значения в результирующей колонке SQL запроса. В таких случаях обычно требуется считывать значение небольшими порциями (*chunks*).

Мы определим средства конфигурирования чтения и записи атрибутов согласно специфическим требованиям механизмов хранения. Введём базовую абстракцию для механизмов — класс *PersistenceMechanism*, и будем использовать полиморфизм для переопределения стандартного отображения. А именно, как показано на рис. 3.3, индикация необходимо-

сти конфигурирования происходит при помощи переопределения в подклассе класса *PersistenceMechanism* (механизм хранения) метода `confirmHasCustomization()`. Этот метод используется для выяснения необходимости выполнения специальных действий во время чтения и записи объекта. Например, для сохранения и загрузки объёмного текстового атрибута в базу данных могут потребоваться хранимые процедуры, которые позволят передавать данные порциями. Это и будут специальные (*customized*) действия.

Сообщение `confirmHasCustomization()`, параметризованное мета-атрибутом объекта, посылается механизму во время отображения, чтобы понять, следует ли использовать стандартную схему обработки для данного атрибута или же требуется вызвать специальную функцию для его загрузки/сохранения. Вся специальная информация хранится в метамодели. Там же определяются специальные методы чтения и записи.

Полезность настраиваемых операций хранения видна, когда стоит задача перевода существующей программной системы на новую архитектуру или технологию. Например, при построении объектно-ориентированного интерфейса к давно существующей реляционной базе данных. В этом случае обычно не удаётся установить общие правила отображения для всех элементов модели данных, поскольку объектная модель не является первичной.

На рис. 3.2 представлены средства для мета-описания сущностей. Этими программными средствами описываются все хранимые классы, их атрибуты и ассоциации. Особый интерес представляет класс *UnidirectionalAssociation* (однонаправленная ассоциация). Чтобы не усложнять модель и процесс отображения (*mapping*), отдельный тип для двунаправленной ассоциации не предусмотрен, и каждое двунаправленное отношение моделируется парой однонаправленных. Для ассоциации задаются такие важные свойства, как кардинальность, тип агрегирования, требование отложенной загрузки (`isProxy`).

На уровне обобщённого хранилища с помощью классов *UnidirectionalAssociation* и *Attribute* происходит установка и чтение связей и атрибутов для отдельных экземпляров. При этом уровень бизнес-логики использует атрибуты и связи непосредственно и не нуждается в особом описании метамодели. Помимо того что метакласс *Class* является контейнером для мета-атрибутов и мета-ассоциаций, он отвечает за создание новых экземпляров и предоставляет информацию об иерархии наследования.

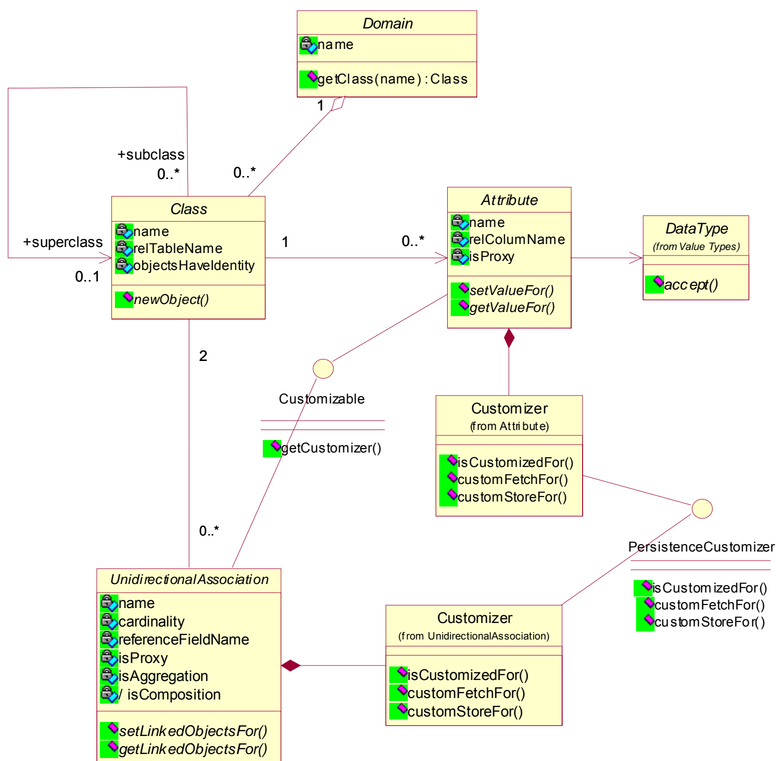


**Рис. 3.2.** Метамодель

## 3.4. Базовые классы сущностей

Для сущностей также необходимо ввести определённый набор базовых и утилитарных типов. Главную роль в хранении объектов приложения играют классы *PersistentObject* и *PersistentID*. Они предоставляют мета-информацию; обеспечивают отображение в реляционное поле свойства идентичности объекта; предоставляют интерфейс для записи, чтения данных и удаления экземпляров хранимых типов. Итак, мета-модель описывает поля данных и отношения объектов класса *PersistentObject* (рис. 3.3).

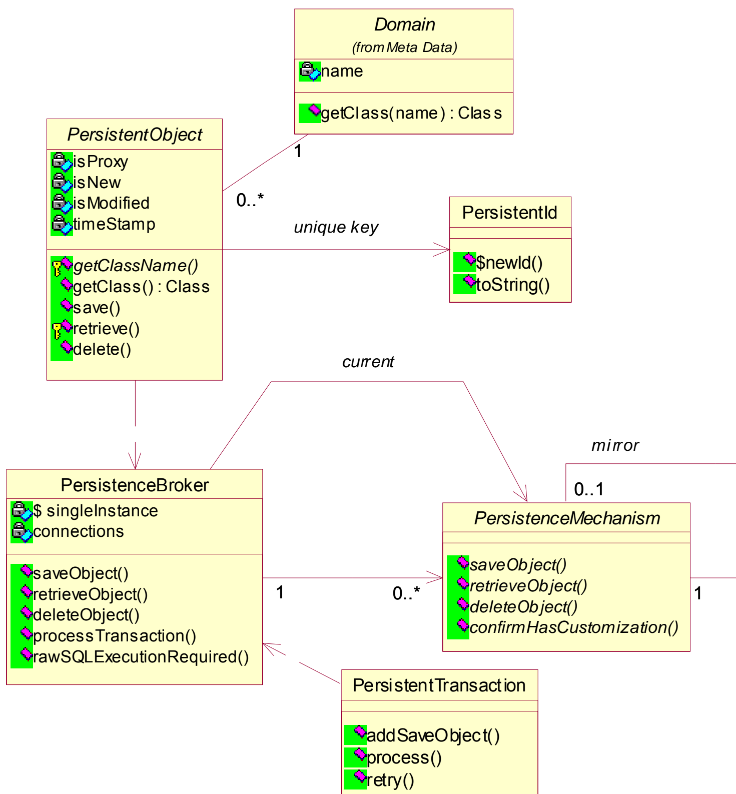


**Рис. 3.3.** Средства хранения

Класс *PersistentObject* является базовым для всех сущностей. Каждому экземпляру *PersistentObject* соответствует объект *PersistentId*, который однозначно идентифицирует сущность в хранилище данных. В интерфейсе *PersistentObject* определены операции `save()`, `retrieve()`, `delete()`, с помощью которых происходит запись, загрузка и удаление объекта. Все классы одной предметной области входят в домен, который представлен классом *Domain*. В домене также агрегирована метамодель для сущностных классов. Получение метаинформации для экземпляра всегда начинается с отправки ему сообщения `getClass()`.

Всё взаимодействие с хранилищами данных происходит через объект класса *PersistenceBroker* (брокер хранения). Этот брокер запросов реализует шаблон проектирования *Singleton* [6], т.е. во время выполнения существует только один его экземпляр. Посредник *PersistenceBroker* делегирует операции хранения текущему механизму и контролирует весь набор имеющихся механизмов хранения. Загрузка, запись, удаление, обработка транзакций (объектов *PersistentTransaction*) происходит только с использованием брокера запросов. Следует обратить особое внимание на сообщение `rawSQLExecutionRequired()`, которое посылается посреднику уровня хранения в случае необходимости исполнения низкоуровневого запроса к реляционной базе данных, например, для настраиваемой загрузки атрибута. Если текущий механизм хранения не поддерживает SQL, то будет сгенерировано исключение. Доступ к реляционным базам данных реализуется в специальном модуле, который содержит класс *QueryExecutor* (исполнитель запросов). Объект именно этого класса будет предоставлен после вызова операции `rawSQLExecutionRequired()`.

Хотя мы говорим об обобщённом хранилище данных, основным носителем информации в нашей модели является реляционная БД с SQL-интерфейсом. И большую часть времени бизнес-приложение должно взаимодействовать именно с этим механизмом. Неудивительно, что на данном уровне предъявляются самые жёсткие требования к производительности. Наличие SQL-запросов в объектно-ориентированном коде является грубейшим нарушением инкапсуляции, но бывают случаи, когда без этого действительно не обойтись [4]. Любой такой исключительный случай должен быть тщательно проанализирован. Иногда удаётся избежать использования SQL путём универсализации подсистемы хранения.

Ниже мы рассмотрим, как механизм хранения и построитель SQL-запросов учитывают конфигурирование записи и чтения значений атрибутов и установки связей (*customization*).

### 3.5. SQL-контроллеры

SQL-контроллерами, или SQL-отображениями, мы будем называть объекты, отвечающие непосредственно за создание SQL-запросов. Более точно, зоной их ответственности является построение запроса, оперирующего данными заданного объекта, группы объектов либо целого класса хранимых сущностей.

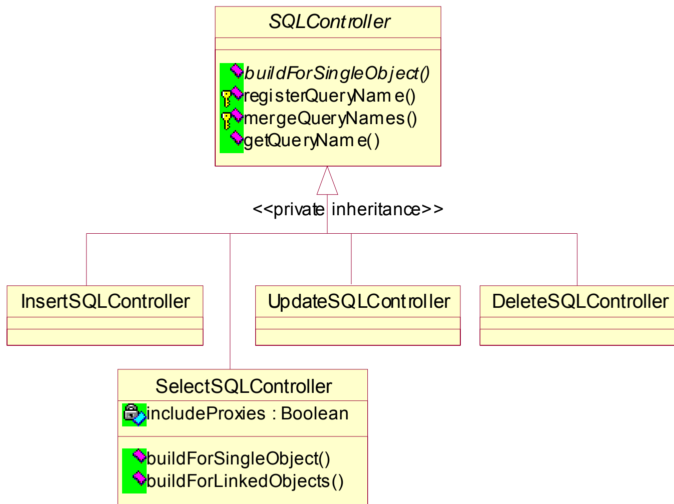


**Рис. 3.4.** SQL контроллеры

Контроллеры используют мета-информацию и правила сопоставления объектной модели со структурой таблиц реляционной базы данных. Упомянутые правила сопоставления и именования лучше выделить в отдельную группу объектов или службу. Мы же рассмотрим упрощённую схему генерации, когда правила жёстко прописаны в коде контроллера.

Автоматически сгенерированный запрос может содержать сотни возвращаемых полей данных, выбранных из нескольких десятков таблиц, в случае SQL-оператора `SELECT`. Аналогично, операторы `INSERT` и `UPDATE` требуют установки большого количества параметров: значений атрибутов объекта и идентификаторов связанных сущностей. Кроме того, для каждой фигурирующей в запросе таблицы возможно более чем одно вхождение, когда между двумя классами существует более одной ассоциации. То, как решаются все эти задачи, мы рассмотрим в п. 0.

### *3.5.1. Абстракция для SQL-выражений*

У каждой реляционной СУБД может быть свой диалект языка SQL. Требуется разумная объектно-ориентированная декомпозиция, позволяющая абстрагироваться от конкретного диалекта и иметь единый интерфейс уровня отображения для всех реляционных хранилищ. С помощью этого интерфейса будут осуществляться полиморфные вызовы каждой из реализаций.

Предлагается использовать библиотеку классов, совокупность которых представляет собой язык SQL. Таким образом, контроллеры SQL смогут работать не с запросами-текстами, а с запросами-объектами. Сконструированный запрос-объект должен быть способен сгенерировать изоморфный объектному представлению SQL-текст. Этот текст будет использоваться для непосредственного обращения к реляционной СУБД.

Описываемая библиотека классов обязана содержать все ключевые абстракции языка SQL, такие как «Таблица», «Запрос», «Упорядочение», «Выражение», «Терм» и прочие.

В качестве примера приведём сильно упрощённую и урезанную объектную модель для SQL-оператора `SELECT`. В данном случае с помощью этого оператора возможно получение новой таблицы данных, состоящей из колонок различных реляционных таблиц. При этом, на получаемые строки накладываются ограничения в виде набора логических условий. Условие формируется из композиций логических связок равенств и неравенств (*термов*). В правых и левых частях термов могут находиться некоторые математические выражения: функции и операторы, использующие в качестве аргументов значения выбираемых полей данных.

Проиллюстрируем необходимость абстракции для SQL различиями некоторых диалектов.

- В СУБД *Oracle* для внешнего связывания таблиц используется знак `(+)` в условии, содержащем внешний ключ, в то время как в СУБД *Access* для тех же целей предоставляется текстовое выражение `OUTER JOIN`.
- В *Oracle* позиции параметров запроса фиксируются строкой `:ParamName`; в интерфейсе ADO используются знаки вопроса `?` и номера позиций.

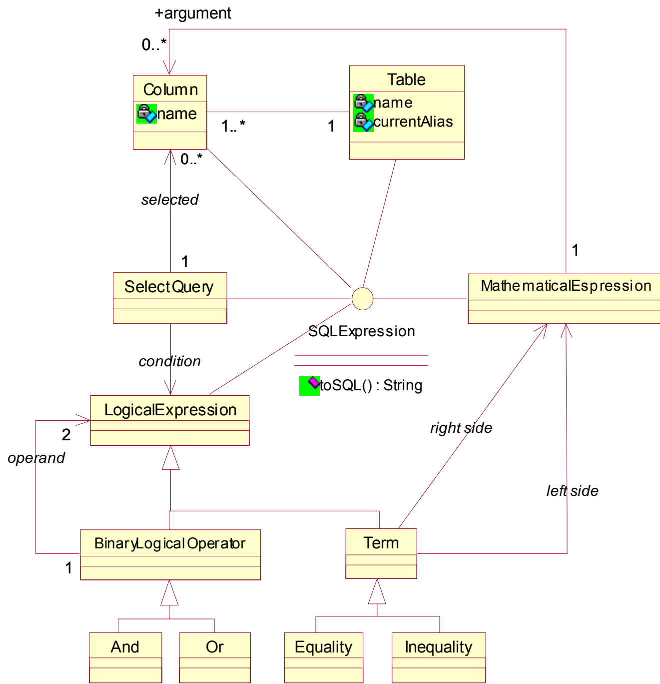


**Рис. 3.5.** Фрагмент объектного SQL

### *3.5.2. Построение по мета-информации*

Входными данными для запроса могут являться условия выборки объектов данного типа, идентификатор объекта, поля которого необходимо обновить в базе данных, определитель порядка возвращаемых строк и прочее. Выходные данные предоставляет только контроллер оператора `SELECT`, и они характеризуют возвращаемые запросом колонки таблиц.

Для работы с входными и выходными данными контроллеры поддерживают две соответствующие карты имён. Карта имён формируется контроллером при построении запроса, и в случае выходных данных для оператора `SELECT` она описывает биекцию псевдонимов возвращаемых запросом колонок и пар <*мета-атрибут класса, мета-ассоциация класса*>.

Уникальные псевдонимы для таблиц и колонок генерируются SQL-контроллером автоматически и в конечном итоге прописываются в SQL-коде запроса:

```
… <Квалификатор колонки/таблицы> as <Псевдоним> …
```

После выполнения запроса механизм хранения «*Реляционная база данных*» использует указанную карту имён для заполнения объектов данными. Создаётся «пустой», неинициализированный объект. Затем с помощью карты устанавливается, под какими псевдонимами возвращаются значения всех атрибутов, и наконец, с помощью правильных мета-атрибутов устанавливается состояние данных объекта.

Не случайно в выходной карте имён используются пары <*мета-атрибут класса, мета-ассоциация класса*>, а не просто <*мета-атрибут класса*>. Дело в том, что одна из задач, решаемых SQL контроллерами, — это минимизация количества запросов к СУБД. Ведь в рамках рассматриваемой нами архитектуры канал взаимодействия приложения с внешней базой данных — самое медленное связующее звено. Гораздо выгоднее, к примеру, получать за один запрос `SELECT` данные двух связанных объектов, чем делать два индивидуальных запроса для каждого из объектов.

Пусть требуется загрузка данных объекта A. При этом SQL-контроллер отвечает за построение запроса, возвращающего не только собственные атрибуты объекта A, но и, рекурсивно, поля объектов, связанных с A экземплярами не-*proxy* ассоциаций кардинальности «1:1» или «1:0..1» (реляционное *внутреннее* и *внешнее* связывание таблиц соответственно). В этом случае второй элемент упорядоченной пары <*мета-атрибут класса, мета-ассоциация класса*> в карте имён будет пустым для собственных атрибутов объекта A. Если же элемент «*мета-ассоциация класса*» в паре не пуст, то «*мета-атрибут*» относится к классу, связанному соответствующей ассоциацией. Все эти данные используются для восстановления состояния целой связки объектов, требуя выполнения лишь одного запроса к реляционной СУБД.

Аналогично ведётся работа с картой имён входных данных. Только в этом случае мета-атрибуты сопоставляются с именами параметров SQL, так называемых *bind*-переменных запроса. Именно *bind*-переменные позволяют параметризовать генерируемые запросы классами, а не конкретными их экземплярами. Т.е. один и тот же запрос можно повторно использовать для целого класса хранимых объектов. Это позволяет эффективно поддерживать кэш запросов и минимизировать расходы на генерацию SQL-текста.

### 3.6. Работа механизма хранения

Механизм хранения «*Реляционная база данных*» тесно взаимодействует с SQL-контроллерами и использует результаты этого взаимодействия — SQL-запросы для взаимной трансляции объектных и реляционных данных. Так, если низкоуровневый интерфейс доступа реляционной СУБД (такой как ADO, JDBC, OCI, OO4O) возвращает в качестве результата запроса итератор, перебирающий строки связанных таблиц, то обобщённое хранилище объектных данных скрывает его за итератором по экземплярам хранимых типов. Т.е. бизнес-приложению, получающему данные, не известно, что именно реляционная база данных является текущим носителем информации. На построении критериев выборки мы остановимся более подробно в п. 3.7.

Рассмотрим работу реляционного механизма хранения в следующих простых случаях:

- сохранение состояния индивидуального объекта;
- загрузка индивидуального объекта, имеющего определённый идентификатор;

**Сохранение состояния объекта** происходит при отправке ему сообщения `save()`. Обработка этого сообщения делегируется сначала единственному в системе *брокеру хранения*, а затем адресуется текущему носителю информации — реляционной СУБД. В зависимости от того, является объект новым или уже сохранённым в БД, механизм «*Реляционная база данных*» инициирует подготовку запроса через `InsertSQLController` или `UpdateSQLController` соответственно. SQL-котроллер, используя мета-данные, подготавливает запрос для сохранения полей объекта и карту имён для *bind*-переменных. Механизм использует имена параметров из карты и устанавливает значения этих параметров в сессии СУБД, используя данные сохраняемого объекта, также получаемые через мета-интерфейс. После чего, запрос переводится в текстовую форму и исполняется.

Затем происходит сохранение настраиваемых атрибутов, которые не покрываются общей схемой отображения объектной модели в структуру реляционных таблиц.

Кроме того, выполняется рекурсивное сохранение агрегированных сущностей, т.е. объектов со связями, являющимися экземплярами межклассовых ассоциаций, для которых установлен признак `isAggregation`.

Для **загрузки объекта** необходимо отправить брокеру хранения сообщение retrieveObject() с указанием идентификатора загружаемой сущности. Сообщение делегируется механизму хранения, в нашем случае —

реляционному. Запрос и карта имён подготавливается утилитой `SelectSQLController`. Механизм «*Реляционная база данных*» с помощью мета-класса создаёт неинициализированный экземпляр, и после выполнения запроса с помощью карты имён и мета-интерфейса устанавливает состояние хранимой в БД сущности. Кроме данных самого объекта, рекурсивно загружаются не-*proxy* связи «1:1» и «1:0..1». Здесь механизм хранения сотрудничает с SQL-контроллером для минимизации количества запросов, проводя необходимую инициализацию связанных объектов.

После загрузки «стандартных» полей данных и связей происходит загрузка настраиваемых полей и связей.

Отметим, что для работы с атрибутами объекта через мета-описание очень удобным оказывается использование вариантного типа данных — объектного аналога `VARIANT` из модели Microsoft COM. Вариантные переменные могут содержать любой из конкретных типов данных, используемых сущностями и отображаемых в систему типов БД. Это позволяет избежать значительных затрат на кодирование приведения типов для каждой операции, вовлекающей мета-объекты.

### 3.7. Критерии выборки

Очень часто требуется выбрать по некоторому критерию и загрузить из хранилища более чем один объект. Например, для отчёта или для получения результатов параметризованного поиска. Нередки и ситуации, когда необходимо удалить определённый набор объектов.

Критерий выбора создаётся программно в терминах логических связок и предикатов, использующих элементы метамодели в качестве аргументов. Например: «все объекты `X` класса `A`, имеющие атрибут `nnn` равный `16.11.1979`, связанные ассоциацией `mmm` с экземпляром класса `B`, у которого атрибут `ppp` больше атрибута `qqq` упомянутого объекта `X`». Таким образом, необходимо реализовать программные абстракции, поддерживающие построение подобных условий.

Критерий направляется брокеру хранения, и задача выполнения соответствующего запроса делегируется текущему механизму. В случае реляционного механизма хранения критерий порождает некоторые условия на языке SQL. Результаты выборки становятся доступны через итератор, перебирающий сущности.

### 3.8. Итераторы и proxy-объекты

Из соображений эффективности механизм, обеспечивающий возможность выборки множества объектов, должен обеспечивать и специальный способ обхода результатов. Курсоры или *итераторы* позволяют обрабатывать объекты по одному, не загружая в оперативную память всю выборку из сотен или даже тысяч экземпляров. В сочетании с разумным кэшированием такой подход обычно оказывается оптимальным, поскольку редко когда требуется обработка или отображение сразу всего множества полученных результатов.

Техника «заместителей» — *proxies* — дополняет использование итераторов и способствует минимизации интенсивности взаимодействия между приложением и носителем данных. «Заместитель» объекта является его «облегчённым» представителем. *Proxy*-объект содержит лишь информацию достаточную для идентификации на уровне приложения и на уровне пользовательского интерфейса. Очень часто proxy-объекты используются для отображения результатов запроса в виде списка, из которого пользователь приложения может выбрать элемент, запросив тем самым более детальную информацию. При этом, через механизм хранения автоматически будет загружен «полновесный» объект [4].

## 4. ЗАКЛЮЧЕНИЕ

Итак, мы рассмотрели некоторые важные для автоматического генератора запросов моменты. Основными результатами являются следующие проектные решения.

- Интеграция с обобщённым уровнем хранения объектных данных.
- Минимизация количества запросов с рекурсивным построением карты имён.
- Средства настраиваемой загрузки и сохранения данных.

Важно, что все излагаемые методы были успешно опробованы автором в процессе реализации промышленной программной системы. Таким образом, и их практическая значимость подтверждена.

## СПИСОК ЛИТЕРАТУРЫ

1. **Кречетов И.В.** Эффективное использование реляционной СУБД в объектно-ориентированном приложении: магистерская диссертация. — Новосибирск, 2003. — 50 с.
2. **Ambler S.W.** Mapping Objects to Relational Databases. — Denver, 2000. — 40 p. — (White Paper / Ronin International; http://www.AmbySoft.com/mappingObjects.pdf).
3. **Буч Г.** Объектно-ориентированный анализ и проектирование с примерами приложений на С++: Пер. с англ. — М.: Бином, 1999. — 560 с.: ил.
4. **Ambler S.W.** The Design of a Robust Persistence Layer For Relational Databases. — Denver, 2000. — 36 p. — (White Paper / Ronin International; http://www.ambysoft.com/persistenceLayer.pdf).
5. **OMG** Unified Modeling Language Specification. — Version 1.3. March 2000. — 1034 p.
6. **Gamma E. et al.** Design Patterns. Elements of Reusable Object-Oriented Software / E. Gamma, R. Helm, R. Johnson, J. Vlissides. — Berkley, California: Addison-Wesley, 1995.

**И.В. Кречетов**

**АВТОМАТИЧЕСКОЕ ПОСТРОЕНИЕ SQL ЗАПРОСОВ ДЛЯ ОБЪЕКТНЫХ ДАННЫХ**

**Препринт**
**118**

Рукопись поступила в редакцию 15.10.04
Редактор З. В. Скок
Рецензент Ю. А. Загорулько

---

Подписано в печать 02.12.04
Формат бумаги 60 × 84 1/16 Объем 1.4 уч.-изд.л., 1.5 п.л.
Тираж 60 экз.

---

ЗАО РИЦ «Прайс-курьер»
630090, г. Новосибирск, пр. Акад. Лаврентьева, 6, тел. (383-2) 34-22-02