Проблемы организации объектно-ориентированного доступа к реляционным базам данных*

К.В. Антипин, В.В. Рубанов

Аннотация. В статье рассматриваются проблемы взаимодействия объектно-ориентированных программ с реляционными базами данных. На основе анализа различных программных продуктов и международных стандартов сделана попытка выделить общие для всех объектно-ориентированных программ особенности долговременного хранения информации. В качестве одного из решений приводится схема трехуровневой архитектуры доступа к хранимой информации Клиент/Посредник/РСУБД, где Посредник играет роль изолирующей прослойки между объектно-ориентированной и реляционной парадигмами.

1. Введение

Объектно-ориентированные (ОО) технологии всё шире внедряются в различные сферы прикладного программирования. Рынок ОО программных продуктов растет с каждым годом. Этот процесс неизбежен и вполне оправдан известными достоинствами ОО подхода:

- надежность программ,
- строгая типизация (классы, подклассы и т. п.),
- фиксация интерфейсов,
- доступ к данным объектов только через их интерфейсы,
- механизм исключительных ситуаций,
- простота проектирования,
- структурированность программы, и как следствие возможность легкой модификации кода.

Одним из главных преимуществ ОО парадигмы над традиционным алгоритмическим программированием является возможность моделирования более сложных и больших задач, что имеет исключительную ценность, поскольку

* Работа поддержана грантами РФФИ № 96-07-89591 и № 97-01-00142.

масштабы и мощность прикладных систем непрерывно растут вслед за увеличением мощности аппаратуры.

При этом большой процент прикладных программ нуждается в долговременном хранении данных, используемых в работе. На основе изучения существующих материалов и программных продуктов можно выделить основные на сегодняшний день технологии хранения, обработки и доступа к данным:

- реляционная модель, получившая наибольшее распространение
- модель объектно-ориентированной СУБД
- СУБД иерархической системы (в виде дерева директорий)
- разнообразные версии гибридных СУБД, в которых делаются попытки объединить преимущества различных подходов.

Таким образом, с одной стороны просматривается четкая тенденция в развитии объектно-ориентированных программ, а с другой накоплено и используется большое число реляционных баз данных, с которыми эти ОО приложения должны взаимодействовать. При этом, говоря о реляционных базах данных, следует отметить, что их программный интерфейс (API) в большинстве случаев разрабатывался для алгоритмических языков программирования и в результате не приспособлен к ОО концепции. Поэтому возникает ряд трудностей при попытке использования API реляционной БД в ОО приложении, рассмотрению которых и посвящен следующий раздел.

2. Хранение объектов в ОО программах

2.1 ОО программирование и реляционные базы данных: проблемы сосуществования

В мире баз данных длительное время развивались и совершенствовались реляционные системы. Создание строгого стандарта языка запросов (*SQL*) явилось одним из наиболее действенных факторов, повлиявшим на появление множественных реализаций РСУБД и их повсеместное внедрение. Реляционная модель получила мощное развитие за долгие годы ее использования и сейчас применяется практически во всех областях, где используются информационные системы. В большинстве своем приложения для РСУБД написаны на алгоритмических языках (*C, Pascal, Fortran*, и т. д.) и для доступа к данным используют библиотеки функций прикладного программного интерфейса (API) данной РСУБД. При попытке применить реляционную технологию хранения данных к ОО программам, возникает ряд несоответствий и даже противоречий. Суть этих противоречий кроется в принципиальном различии двух подходов: реляционного и объектно-ориентированного.

ОО модель оперирует такими аспектами как данные (информационные элементы), содержащиеся в объектах, поведение объектов, представленное

набором методов, и взаимосвязи различного типа между классами этих объектов. Тогда как РСУБД оперирует только данными и предназначена для хранения собственно информационных массивов. При этом для объектной модели характерно наличие таких свойств, как инкапсуляция, наследование и полиморфизм. Объекты в ОО модели органично связаны в некоторую схему, которая находится в довольно простом соответствии с жизненной задачей приложения. Реляционная модель направлена, прежде всего, на именно управление большими объемами абстрактных данных некоторых предопределенных элементарных типов. Все данные хранятся в виде двумерных таблиц (в настоящее время существуют также и многомерные РБД) и в качестве механизма обеспечения целостности данных используются первичные и внешние ключи. Доступ к данным осуществляется путем выполнения SQL запросов, при этом в качестве объекта действия выступает одна или несколько таблиц.

Рассмотрим трудности, возникающие при попытке отображения ОО модели на реляционную, то есть при решении задачи хранения объектов в реляционной среде.

1. ***Инкапсуляция.*** Реляционные БД не поддерживают средств ограничения доступа, кроме паролей и пользователей. Однако прикладная программа обычно выполняется при одном уровне привилегий и поэтому все приватные данные, хранящиеся в БД, становятся общедоступными, а попытки обеспечить инкапсуляцию путем введения различных представлений (views) с определенными правами доступа ведут к существенному усложнению схемы БД и программного кода.
2. ***Наследование.*** Так как в РБД хранятся только данные объектов, то для корректного их воссоздания нужно хранить еще и информацию об их классах. Иерархия классов может быть представлена путем создания отдельных таблиц (*таблиц класса*) для каждого элемента этой иерархии. При этом для обеспечения корректного выполнения запросов записи для объекта необходимо хранить и во всех таблицах родительских классов и при обработке данного объекта надо следить за правильным использованием всех этих родительских таблиц.
3. ***Полиморфизм.*** Поддержка данного механизма в РБД полностью отсутствует, и чтобы обеспечить корректную реконструкцию объектов, нужно следить за наличием достаточной дополнительной информации в таблицах класса.

Итак, мы видим, что обеспечение базовых механизмов ОО модели в реляционной БД связано с рядом трудностей, связанных, прежде всего, с необходимостью решать задачу корректного отображения иерархии объектов на реляционную структуру, что зачастую является очень нетривиальным делом. Кроме того, сам прикладной интерфейс РСУБД не соответствует объектным принципам и его использование вкупе с необходимостью хранения объекта в нескольких различных таблицах заметно повышает сложность кода и как следствие нарушает его структурированность, существенно усложняет процесс отладки. При этом SQL-код в явном виде используется в теле

приложения, что делает программы негибкими и непереносимыми, так как появляется зависимость приложения от используемой схемы РСУБД (структура таблиц, связи между ними, поля таблиц) и зависимость кода программы от конкретной РСУБД (производителя, а возможно, и версии).

Таким образом, попытка непосредственного использования традиционных *API* реляционной СУБД в ОО среде клиента нивелирует преимущества ОО концепции.

В настоящий момент в мире разработано огромное число информационных систем на базе реляционных СУБД. И проблема "сращивания" ОО приложений с РСУБД представляется весьма актуальной из-за наличия большого числа "наследуемых систем" ("*legacy systems*"). В то же время, в связи с увеличением интереса к ОО программированию, в последнее время получают все большее распространение *объектно-ориентированные системы управления базами данных (ООСУБД)*. Технология ООСУБД явилась альтернативой реляционным базам данных и отчасти была призвана разрешить те несоответствия ОО и реляционного подходов, о которых говорилось выше. Однако её использование подразумевает создание информационной системы «с нуля».

Более подробному анализу преимуществ, недостатков и перспектив использования реляционных и объектно-ориентированных СУБД и посвящен следующий подраздел.

2.2 Реляционные и объектно-ориентированные СУБД: современное состояние и перспективы

Как было отмечено, в настоящее время можно четко выделить два основных направления в хранении данных приложения: использование *объектных* и *реляционных* баз данных. При этом в обоих случаях имеются свои преимущества и недостатки. И решение в пользу того или иного направления главным образом определяется самой решаемой задачей.

Реляционный подход, будучи хорошо изучен, развит и стандартизован в результате многолетней практики использования, хорошо подходит для задач, оперирующих большими объемами данных, когда критическими являются скорость обработки и возможность формирования сложных составных запросов на выборку данных. Однако для успешного применения *РСУБД* в совокупности с ОО программами, необходимо иметь специальную архитектуру и технологию хранения объектов в *РСУБД*. Также нельзя сбрасывать со счетов проблему унаследованных систем – факта существования огромного числа действующих, годами разрабатывавшихся реляционных БД. Революционное внедрение новых технологий (например, *ООСУБД*) потребовало бы колоссальных вложений на перепроектирование существующих систем и на трансляцию накопленных данных в новую БД – непростую, а в некоторых случаях крайне трудно решаемую задачу, в силу больших размеров и сложности исходной базы. И все это в условиях необходимости непрерывного доступа к транслируемым данным для нормального функционирования

предприятия. *ООСУБД*, несомненно, имеют свою область применения. Существует класс задач (*CAD applications, CASE tools* и другие), где применение *РСУБД* не вполне целесообразно, поскольку они не смогут эффективно работать, тогда, как *ООСУБД* исключительно хорошо удовлетворяют требованиям задачи. Например, в ОО системах автоматизированного проектирования обычно строится модель из сложных, тесно взаимосвязанных объектов. Дизайнер обычно долго манипулирует системой из таких объектов, после чего вся конфигурация должна быть сохранена. Если такое приложение базируется на РСУБД, для создания столь сложных и тесно взаимосвязанных объектов потребуется производить объединение многих таблиц и выполнять много операций для "синтеза" окончательного объекта, что неизбежно скажется на производительности. В этом случае удобнее и быстрее сохранить конфигурацию как объект.

Отметим, что *ООСУБД* являются развивающейся технологией, для которой еще не существует официального промышленного стандарта, однако работы активно ведутся. В 1993 году была создана первая версия стандарта на *ООСУБД* – ***ODMG-93.*** Автором является рабочая группа по объектным базам данных (*Object Database Management Group, с 28.04.98 переименована в Object Data Management Group*), в состав которой входят наиболее влиятельные в области информационных технологий фирмы. После чего последовала большая работа по исправлению и доработке, которая в результате воплотилась уже в более детальный стандарт ***ODMG 2.0***. Таким образом, просматривается явная тенденция в сторону развития объектно-ориентированных БД. Однако на сегодняшний день и в ближайшем будущем, по-видимому, будут превалировать *РСУБД*, хотя бы в силу человеческой инертности. При этом следует заметить, что реляционные базы данных выполняют строго предписанную им функцию – хранение данных, и с этой задачей они прекрасно справляются, что вполне оправдывает их использование в любых программах. Проблема же рассогласования двух парадигм представляется более общей. Как пишет автор статьи [8], это проблема сосуществования одной модели с другой несколько сродни идее инкапсуляции, положенной в основу ОО парадигмы. Следует изолировать данные (информацию), которые статичны по своей природе и функции (алгоритмы), непрерывно меняющиеся под влиянием переменных внешних условий. При этом рождается концепция изолирования доступа к данным со стороны программы и создания функционального интерфейса, такого, чтобы программа могла обращаться к этим данным лишь в терминах некоторых функций, несущих определённый смысл в предметной области своей задачи. Это позволит произвольно и прозрачно для приложения изменять представление данных во внешнем хранилище (РСУБД), не модифицируя код приложения. На этой идее базируется трехуровневая архитектура организации хранения данных, которая будет подробней рассмотрена в последней главе. Вопросам же реализации такой системы посвящается статья «Среда ParJava для разработки SPDM-программ для однородных и неоднородных сетей JavaVM».

3. Особенности проблемы долговременного хранения информации для ОО программ

В данном разделе остановимся на рассмотрении некоторых конкретных деталей организации долговременного хранения информации в объектно-ориентированных программах. Набор концепций и правил, на которых строится ОО приложение, порождает ряд специфических особенностей проблемы хранения информации в ОО системе. Если сравнить исследования, проводившиеся группами, которые разрабатывали различные технологии долговременного хранения информации (*ООСУБД*, *ОРСУБД*, *CORBA Persistent Object Service*, 3-х уровневая архитектура клиент/сервер приложения/*РСУБД*), то легко увидеть очень много общего, как в самой проблематике этих исследований, так и в подходах к ее разрешению. Попробуем на примере нескольких работ ([19] – [22]) осветить возникающие вопросы и предложенные пути их решения.

3.1 Долговременное хранение объектов. Механизмы сохранения и восстановления состояний объектов.

Начнем с терминологии, применяемой для описания функций хранения информации в ОО системах. Общепринятым и распространенным стало понятие *хранимости* и *хранимого объекта или класса (persistency и persistent object (class))*

Хранимость – особое свойство объектов, состоящее в том, что объект может сделать "снимок" своего состояния и сохранить его в некотором хранилище (например, ООСУБД), после чего он может закончить свое существование и быть восстановлен в следующем сеансе работы программы. При этом такой объект, обладающий свойством хранимости, называется *хранимым*.

В дальнейшем можно создать другой объект того же класса (или любого родительского класса исходного объекта) и автоматически проинициализировать его атрибуты, используя сохраненное в базе данных состояние [22]. В результате мы получаем возможность хранить состояние объектов дольше, чем время жизни самих объектов, что фактически означает хранение необходимой клиенту информации. При этом, поскольку внутренние структуры данных являются "собственностью" объекта и скрыты от непосредственного доступа извне, хранимый объект должен предоставлять некоторые методы, с помощью которых можно сохранить и восстановить его состояние. Более того, в сохраняемое состояние объекта могут входить не все его внутренние переменные, а лишь некоторые, о чем "знает" только сам объект. [18], [19]. Реализуется эта идея обычно с помощью наследования объектом от некоторого абстрактного класса или интерфейса, содержащего соответствующие методы сохранения, которые могут реализовываться по-разному в объектах разных классов.

Альтернативой такой техники сохранения/восстановления состояния объектов могла бы быть некоторая внешняя система, отвечающая за хранимость и имеющая эксклюзивное право доступа к внутренним переменным любого объекта. В такой схеме потенциально можно было бы сохранять/восстанавливать состояние любого объекта. В документе [21] представлено два приема сохранения состояния объекта:

- реализация хранимым-классом хранимого интерфейса;
- сохранение объекта извне, когда для каждого типа внутренних данных объекта существует метод, "умеющий" сохранять переменные этого типа.

Однако второй подход, более свойственный алгоритмическим программам, нарушает идею инкапсуляции данных со всеми вытекающими отсюда негативными последствиями. По этой причине первая схема организации долговременного хранения в ОО приложениях является общепризнанной.

ОО языки поддерживают концепцию уникальной *объектной ссылки (OID)* - идентификатора, которым обладает каждый объект программы. Не вдаваясь в особенности присваивания этой ссылки, отметим, что всякому объекту такой идентификатор присваивается только один раз при его создании и не может изменяться на протяжении всей жизни объекта. Когда состояние объекта сохраняется в базе данных, этому состоянию присваивается уникальный в пространстве всех хранимых состояний идентификатор (отличный от *OID* объекта) [22]. Будем в дальнейшем называть этот идентификатор *POID (Persistent Object ID)*.

Противоположность хранимому объекту – *переходный объект* (*transient object*). Следует отметить, что разные авторы вкладывают в это понятие несколько различный смысл. Например, в [19] переходный объект – это объект, который до текущего момента времени не взаимодействовал с базой данных (не сохранялся и не восстанавливался) и существует только в памяти приложения. Однако в этой интерпретации переходный объект может стать хранимым объектом в будущем, если он принадлежит хранимому классу. С точки зрения же авторов [22] переходный объект - это объект, который существует только в памяти и не может стать хранимым. Аналогично, в работе [21] определяется понятие сохраняемой и переходной памяти для хранимых и переходных объектов соответственно.

3.2 Состояние объекта, межобъектные связи. Проблема поддержки целостности базы данных

Теперь поговорим о состоянии объекта и поясним, что понимается под сохранением состояния объекта. В общем случае объект может содержать в себе скалярные и векторные переменные элементарных и составных типов данного языка, а также ссылки на другие объекты, векторы объектов и т. д.

Безусловно, в состояние объекта входят значения его внутренних переменных, но, очевидно, что и состояния включаемых объектов (объектов, на которые есть ссылки) также определяют состояние нашего объекта. Хотя некоторые авторы склонны называть состоянием объекта набор текущих значений его внутренних переменных, исключая ссылки на объекты. Для того, чтобы охарактеризовать эти ссылки, они определяют понятие *зависимостей* [22] для данного объекта как набор всех объектов, на которые ссылается наш объект, а также понятие *транзитивного замыкания* (*Transitive closure of dependencies*) - набора всех объектов, достижимых из данного объекта по ссылкам на всех уровнях вложенности. Тогда сохранить состояние объекта - означает сохранить лишь его внутренние переменные. Здесь мы близко подходим к проблеме хранения и представления связей в модели классов задачи. В связи с этим возникает следующий ключевой вопрос, – какую часть работы по "поддержке" межклассовых связей должна взять на себя база данных, а какую следует переложить на приложение. От решения этого вопроса во многом зависит функциональность базы данных и логика всех операций обработки данных (`update, insert, delete, ...`).

Поддержка целостности является важной функцией системы хранения информации, независимо от того, какая технология при этом используется (*РСУБД*, *ООСУБД* или архитектуры с промежуточным слоем). При этом может обеспечиваться как структурная целостность данных (чтобы в БД не было ссылок на несуществующие объекты или на объекты несоответствующего типа, зацикливание ссылок, и т. д.), так и логическая (аналог концепции правила "*rule*" или "*constraint*" в реляционных базах данных), с помощью которых можно гарантировать выполнение некоторых семантических ограничений данной задачи [14]. Например, в [14] описывается схема объектной базы данных, которая на основе схемы классовой модели сама следит за соответствием взаимных ссылок экземпляров классов, в ней хранящихся. Как правило, все *ООСУБД* берут на себя такую проверку соответствия[21]. Те же принципы предложены в прототипе стандарта *ODMG 2.0*. По-разному можно решать вопрос внесения изменений в состояние объекта, существующего в окружении связанных с ним объектов. Можно выполнять автоматическое обновление всего этого окружения (в том числе связей) при попытке модификации любого из входящих в нее объектов. Можно считать, что приложение обязано само сохранять состояния связанных объектов оболочки при модификации некоторого из них. Например, в работе [19] "ссылочные объекты" явно подгружаются и обновляются приложением по мере необходимости. В этой модели существуют три вида объектов в ОО приложении, между которыми возможны взаимные переходы в процессе работы:

- *переходный объект*, который не взаимодействовал с базой данных и существует только в памяти программы;

- *хранимый объект,* который взаимодействовал в некоторой транзакции с базой данных, и в ней хранится некоторое состояние этого объекта. В кэше клиента есть ассоциация между данным объектом и этим состоянием в базе данных;
- *теневой объект (shadow object)*, "частично" загруженный в память из базы данных, когда некоторый хранимый-объект имеет ссылку на него. Конкретные переменные объекта не загружены в память. Чтобы объект стал хранимым, нужно явно вызвать операцию загрузки его в память.

На рис. 1 приведена диаграмма состояний объекта (стрелками обозначены переходы по тем или иным событиям).

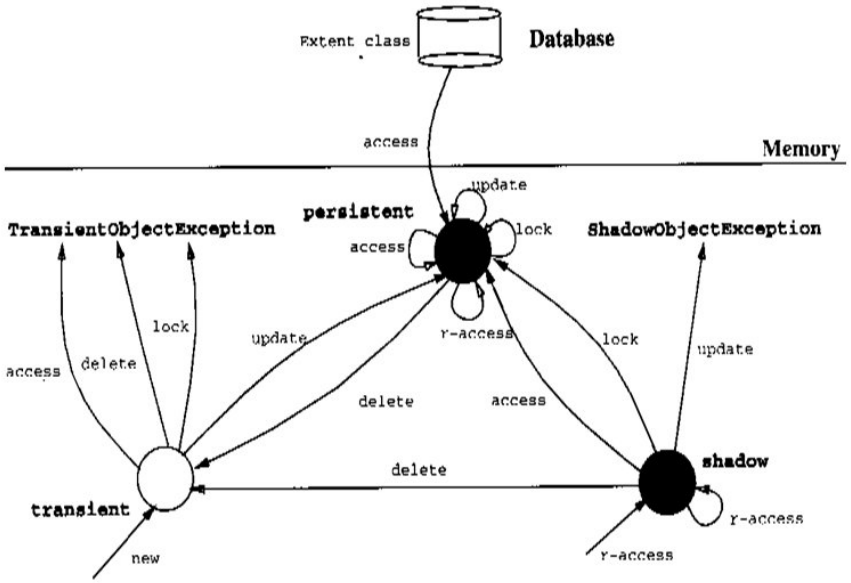

Рис. 1. Диаграмма состояний объекта

Для упрощения этой задачи многие системы поддерживают понятие кластера (*cluster* в *COOL*, *group* в *SOM*), в рамках которого отслеживаются все взаимосвязи между объектами[14]. Кластер становится единицей обмена между хранилищем и программой (при записи в базу и при чтении из нее)[21]. Можно определить некоторый оптимальный размер такого кластера, чтобы минимизировать потери на бесполезную перекачку объектов, которые не используются программой в данной транзакции.

3.3 Варианты техники доступа к хранимым объектам в базе данных. Поиск нужного состояния, язык запросов

Немаловажной проблемой в задаче долговременного хранения данных для ОО систем является вопрос организации доступа к данным со стороны ОО программ. Необходимо продумать эффективную технику нахождения в базе данных нужных состояний объектов, что является особенно актуальным в задачах, требующих обработки больших объемов данных. В реляционной модели для этих целей используется специальный язык запросов *SQL*, позволяющий формировать разнообразные и разносторонние запросы поиска нужных данных. Что касается ОО баз данных, в них применяются некоторые из следующих механизмов получения нужных объектов[14]:

1. Доступ к хранимым объектам посредством так называемой "точки входа" – некоторого выделенного объекта в БД, имеющего уникальное имя.
2. Доступ по объектной ссылке. Через точку входа можно проинициализировать некоторый объект в памяти, после чего через содержащиеся в нем объектные ссылки можно получить доступ к целой группе других хранимых состояний [21]. Эта техника обращения к объектам, хранящимся в базе данных, применяется в модели "хранимой" и "переходной" памяти ООСУБД ObjectStore. Определение точек входа для конкретного случая входит в задачи программиста, пишущего приложение.
3. Доступ к объекту через уникальный POID состояния (см. пункт 1).
4. Доступ с помощью языка запросов. Стандарт ODMG 2.0. определяет для этих целей специальный язык запросов OQL (Object Query Language).

В трехуровневых архитектурах (ОО клиент/Посредник/*РСУБД* сервер) в промежуточном слое Посредника обычно производится преобразование запросов клиента, сформулированных в объектных терминах клиента, в SQL-запросы к реляционному серверу. Обрисуем на примере одной реализации такой системы ([19]) возможные способы получения из базы данных тех или иных состояний. В работе [19] определяются понятия `Proper` и `Transitive Extent` (сегментов) – как набор всех экземпляров данного класса и всех экземпляров данного класса и всех его подклассов соответственно. В процессе работы создается объект класса `Extent` с методами `Proper(String class_name)` и `All(String class_name)`, возвращающими `Proper` и `Transitive` сегменты класса `class_name` соответственно. Эти сегменты, в свою очередь, являются объектами, обладающими методом `Where (String where_clause),` который необходим для того, чтобы отфильтровывать из полученного множества нужные состояния по некоторому критерию. Также имеется специальный метод `getTableNameforClass` объекта `Database`, позволяющий получить имя таблицы в реляционной базе данных, в которой хранятся состояния экземпляров данного класса, после чего использовать это имя в методе `Where`. Также есть возможность писать запросы с объединением таблиц, получая в процессе работы информацию о таблицах, хранящих нужные данные, от Посредника во время работы.

4. Трехуровневая архитектура системы долговременного хранения информации для ОО программ. Связь ОО клиента с реляционной базой данных

Рассмотрим вопросы реализации трехуровневой системы хранения информации *ОО клиент/Посредник/РСУБД,* в которой Посредник обеспечивает прозрачное согласование объектно-ориентированной и реляционных парадигм. При этом клиенту кажется, что он работает с объектно-ориентированной БД, а *РСУБД* в свою очередь получает стандартные SQL-запросы. Одной из наиболее продвинутых и подробно документированных реализаций является трехуровневая архитектура ОО клиент/Посредник/*РСУБД* - в сервере компании *O2-technology* под названием *JRB (Java Relational Binding)* [19]. Ориентированность на три уровня является достаточно общей тенденцией в информационных системах с долговременным хранением информации. И это обусловлено стремлением разграничить две функционально различные части: ОО приложение, решающее специальную задачу своей предметной области, и хранилище информации, в которое обычно стараются заложить такие качества как максимальная эффективность и удобство хранения данных и доступа к ним. Практически в любых моделях таких программных систем существует некое промежуточное звено, предоставляющее адекватный интерфейс той и другой стороне и инкапсулирующее в себе все несоответствия, возникающие на стыке двух подсистем. Будем в дальнейшем называть этот промежуточный компонент трехуровневой архитектуры Посредником.

3.4 Описание одной из реализаций трехуровневой системы

Для более четкого понимания трехуровневой архитектуры рассмотрим в качестве примера подробное описание одной из реализаций - систему *JRB*. В ней *Java*-приложение, взаимодействуя с Посредником через интерфейс специального пакета `COM.unidata.jb.api`, может сохранять состояние своих объектов в реляционной базе данных (*Oracle, Sybase*). Для этого приложению нужно предварительно импортировать в базу данных схему используемых *Java*-классов с помощью утилит, выполняемых в период разработки программы. В результате чего создаются таблицы для хранения объектов этих классов и генерируются некоторые исходные тексты *Java*-классов, выполняющих функции сохранения/восстановления объектов этих классов в базе данных. Код клиента должен компилироваться с включением этих классов.

В работах [20] и [21] предложены схемы разработки, где не требуется включения никакого внешнего кода в тело приложения, получаемого при импортировании. Сохраненные состояния могут быть восстановлены в объектах соответствующих классов. *API JRB* включает в себя специальные классы, с помощью которых можно получать то или иное подмножество хранимых состояний объектов

данного класса. Методы этих классов позволяют формировать разнообразные критерии выборки в запросах. Сохранение/восстановление объектов, на которые ссылается данный объект, производится явным запросом к Посреднику. При этом поддерживается концепция "частично" загруженных - теневых объектов (shadow objects). Также возможно сохранение в базе данных массивов произвольного типа (в т. ч., массива объектов). Для обработки больших массивов объектов используется специальная техника загрузки их в память, позволяющая сократить накладные расходы на выделение большого объема памяти под теневые элементы этих массивов. На стороне клиента производится кэширование состояний, получаемых из базы данных.

Для того чтобы трехуровневая архитектура допускала одновременную работу нескольких приложений, которые принципиально могут обратиться к одному и тому же состоянию в базе данных с запросом на чтение или на запись, необходимо организовать определенную схему разграничения доступа. JRB использует для этих целей классическую концепцию транзакции. По умолчанию, когда приложение открывает базу данных и считывает из нее некоторое состояние, оно работает в режиме `read-only`. Транзакция открывается методом `begin()` класса `Transaction` и может быть завершена вызовом одного из методов: `commit(), validate()` или `abort()`. В *JRB* существует три типа блокировки состояний объектов в базе данных: `shared lock, update lock, exclusive lock`. При обращении в режиме `read-only` объекты никак не блокируются. Когда приложение считывает некоторое состояние, на это состояние накладывается `shared lock`. При этом на одно и то же состояние может быть наложено одновременно несколько блокировок `shared lock`. Но как только приложение вызывает метод `update()` или `markModify()`, на соответствующее состояние накладывается `update lock`. На всякое состояние может быть наложена только одна блокировка типа `update lock`. `Update lock` гарантирует, что после снятия всех прочих блокировок приложение получит `exclusive lock` и тогда изменение состояния объекта будет зафиксировано в базе данных. Однако наличие `update lock` еще не исключает возможности возникновения ситуации `deadlock`. Если приложения заходят в `deadlock`, они должны через некоторый таймаут снять все блокировки и повторить операции снова через случайное время. Для того чтобы гарантировать невозможность возникновения ситуации `deadlock`, предусмотрен метод `lock()`, который сразу пытается получить `exclusive lock` на данное состояние.

5. Заключение

Таким образом, мы рассмотрели проблему организации доступа к РСУБД из объектно-ориентированных программ. В результате было выявлено, что наиболее приемлемой архитектурой, которая могла бы обеспечить корректное разделение

двух подходов, является трехуровневая архитектура, которая с одной стороны эмулирует для ОО приложения интерфейс ООСУБД, а с другой стороны взаимодействует с РСУБД через обычный SQL интерфейс.

Заметим, что трехуровневая архитектура вызывает активный интерес со стороны различных компаний-разработчиков ПО. В применении к хранению данных в БД можно выделить несколько направлений в реализации трехуровневой архитектуры. В первую очередь это трехуровневая организация JDBC интерфейсов. К этой группе относятся такие продукты, как «Data Broker» производства I-Kinetic, «dbAnyWhere» производства Symantec, «jConnect» производства Sybase и др. Вторая группа представлена реализациями *сервиса долговременного хранения* в рамках архитектуры CORBA. К ней, например, относится «Persistent Object Manager» компании Secant. И, наконец, к третьей группе можно отнести различные вариации по реализации ODMG интерфейса, которые часто включают поддержку также и других стандартов (CORBA, RMI, UML modeling). Здесь можно отметить «CocoBase» компании THOUGHT и «Enterprise Object Framework» производства NeXT Computing. К комплексным трехуровневым решениям относится также «JaguarCTS» компании Sybase.

При этом особенно актуальной становится система, реализующая трехуровневую архитектуру и обеспечивающая решение задач как прямого, так и обратного проектирования прикладных программных систем на основе РСУБД.

Литература

1. “Why Use an ODBMS?” POET technical reference. http://www.poet.com/t_oovsre.htm
2. “The Common Object Request Broker: Architecture and Specification”. OMG Document, Revision 2.0, July 1995.
3. “CORBA services: Common Object Service Specification”. OMG Document 95-3-31, 1995. Updated: November 22, 1996.
4. “Persistent Object Service specification”. OMG Document 94-10-7, 1994.
5. "Mapping Objects to Relational Databases". Ambysoft White Paper, Oct. 1997. http://www.ambysoft.com/bibliography.html
6. "Integrating Objects with *RDBMS*". Gemstone System White Paper Inc.,1998
7. Kyle Brown Bruce G. Whitenack "Crossing Chasms: A Pattern Language for Object-*RDBMS* Integration". Knowledge Systems Corp. White Paper.
8. Robert C. Martin "Storing Objects in a Database". Usenet Conference comp.database.object, Jul 1997
9. "Object/Relational Mapping Concepts". ONTOS*Integrator White Paper, http://www.ontos.com/mapcon.htm
10. Kyle Brown Bruce G. Whitenack "Mapping Objects to Tables. A Pattern Language". Knowledge Systems Corp. White Paper.
11. Jan Kleindienst, Frantisek Plasil, Petr Tuma. “Lessons Learned from Implementing the CORBA Persistent Object Service”. OOPSLA’96 conference proceedings, San Jose, California, October 6-10, 1996.
12. Jaromir Adamec, Michael Grof, Jan Kleindienst. “Supporting Interoperability in CORBA via Object Services”. Tech. Report 114. Charles University Prague, Department of Software Engineering, October 1995.

13. J. Kleindienst, F. Plasil, P. Tuma. “Implementing CORBA Persistent Service”, TR 117, Charles University Prague, Department of Software Engineering, 1995.
14. "Developing OO Applications for Relational DBMS’s". Cayenne Software Inc. White Paper, 1997-1998
15. “The Secant Persistent Object Manager”. Object Magazine, May 1997.
16. G. Booch «Object-Oriented Design». Behjamin/Cummings Publishing Company, 1992
17. Loomis Mary E. S «Object and Relational Technology: Can they cooperate?» Journal of OO programming, 1992
18. Loomis Mary E. S «Object and SQL: Accessing Relational Databases» Journal of OO programming, 1992
19. Frank M., "Object-Relational Hybrids". DBMS Magazine, 1996
20. Frost Stuart "Modeling for the *RDBMS* legacy". Object Magazine, Sep 1994
21. G. McFarlay & A. Rudmik "A Critical Review/Technology Assessment ". OODBMS Sep. 1993
22. David Tothwell "Java Object Persistence". Technical Overview, Sep. 1996