Объектно-ориентированное окружение, обеспечивающее доступ к реляционным СУБД*

В.П. Иванников, С.С. Гайсарян, К.В. Антипин, В.В. Рубанов

Аннотация. В статье дается описание объектно-ориентированной (ОО) среды программирования, позволяющей организовать прозрачный доступ к реляционной БД из ОО программы. Дается определение необходимых терминов, описание объектной модели среды, языка построения схем БД и языка запросов, с помощью которого осуществляется поиск и обработка хранимых объектов. Излагаются общее описание алгоритмов работы и этапы разработки целевого ОО приложения. Описанная архитектура базируется на стандарте для объектно-ориентированных СУБД *ODMG 2.0* и реализует некоторое его подмножество. На базе изложенного описания будет реализован прототип трехуровневой системы для ОО программирования задач с хранением информации в РСУБД.

1. Введение

В статье приводится описание схемы и принципов работы объектно-ориентированной среды доступа к реляционной СУБД, основанной на стандарте для объектно-ориентированных СУБД – ***ODMG 2.0.*** Как правило, всякая ОО СУБД состоит из двух четко выраженных подсистем: *объектно-ориентированной среды* (ООС), в которой производится программирование прикладной задачи, и подсистемы, обеспечивающей хранение данных *(Сервиса долговременного хранения (СДХ)-Persistency Service).*

Первая подсистема (ООС) реализуется как библиотека классов, обеспечивающих программисту ОО интерфейс к подсистеме хранения, в который входит базовая функциональность ООСУБД: классы, поддерживающие пользовательские сессии, транзакции, хранение объектов в БД, язык запросов и т. д.

Вторая подсистема (СДХ) непосредственно связана с хранением информации и отвечает за представление данных, предоставляя интерфейс, через который классы первой подсистемы могут выполнять все необходимые для хранения объектов операции.

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

Вследствие такого разделения, есть возможность использования различных СДХ для одной и той же ООС прозрачным для прикладного программиста образом. Обе указанные подсистемы могут быть созданы по отдельности и связаны через интерфейс СДХ.

Разрабатываемая система укладывается в эту общую схему. Заметим, однако, что под ООС часто подразумевается собственно интерфейс, предоставляемый прикладному программисту для использования при написании приложения, а под СДХ – интерфейсы более низкого уровня для связи с ядром ООБД (в нашем случае с РСУБД). В таком аспекте систему можно рассматривать так, как она показана на рис. 1.

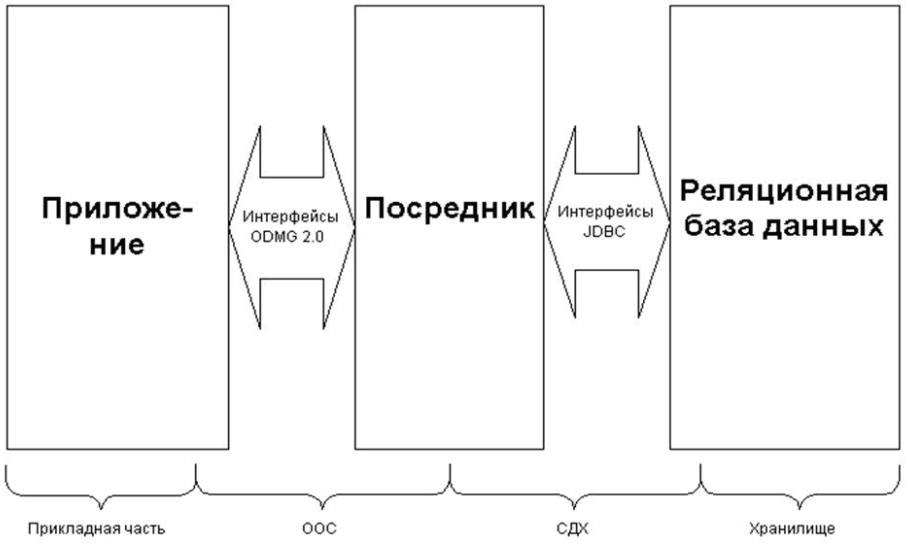

Рис. 1. Архитектура системы долговременного хранения объектов.

Здесь ООС представлена интерфейсами, описанными в стандарте ODMG 2.0 и реализуется в виде упаковки (package) Java и набора системных утилит. А СДХ, в свою очередь, представлен в нашей системе интерфейсами Java Database Connectivity (JDBC) и реализуется соответствующими драйверами и собственно РСУБД. В данной статье мы остановимся на описании именно ООС, которая будет использоваться при программировании и выполнении ОО приложений.

Данная среда, как уже отмечалось, соответствует спецификации интерфейсов, описанной в стандарте *ODMG 2.0.* Этот стандарт специфицирует *объектную модель (ОМ),* которая определяет, каким образом организуется долговременное хранение в ОО приложении. То есть вопросы взаимосвязи и взаимодействия *хранимых объектов (Persistent objects -PO)*, их именования, идентификации и

т.д., а также архитектуру классов, с помощью которых приложение организует хранение, выборку, обработку *PO*. В прототипе системы будут реализованы подмножества объектной модели ODMG, языка определения объектов (ODL) и языка запросов (OQL) в рамках стандарта ODMG 2.0. Указанные подмножества описываются в разделах 2, 3 и 4 соответственно.

2. Объектная модель

2.1 Основные понятия и термины

Раздел содержит описание необходимой для дальнейшего изложения терминологии, определяя *объектную модель*, принятую ODMG в качестве стандарта. Данная модель специфицирует конструкции, которые должны поддерживаться любой ООСУБД:

- Базовыми примитивами являются ***объект*** и ***литерал***. Каждый объект обладает уникальным идентификатором (OID). Литерал не имеет идентификатора.
- ***Состояние*** объекта определяется как совокупность значений его **свойств**. Свойства делятся на ***атрибуты*** и ***связи***.
- ***Поведение*** объекта определяется набором методов этого объекта.
- ***Тип*** объединяет два понятия интерфейса и класса.
- ***Интерфейс*** специфицирует только абстрактное поведение своих экземпляров (без описания состояния).
- ***Класс*** специфицирует абстрактное поведение и абстрактное состояние своих экземпляров.

В БД хранятся объекты, доступ к которым, вообще говоря, может осуществляться из нескольких различных приложений. БД основывается на ***схеме***, написанной на специальном декларативном языке ***ODL*** *(Object Definition Language)*. БД содержит экземпляры типов (объекты), описанных в данной ***схеме***.

Тип обладает внешней ***спецификацией*** и одной или несколькими ***реализациями***. ***Спецификация*** определяет внешние характеристики типа, т. е. характеристики, которые видны пользователям этого типа, а именно:

- атрибуты или переменные состояния (если они есть), значения которых можно получить для каждого экземпляра типа.
- методы, присущие экземплярам типа.
- исключения (exceptions), которые могут возникнуть в процессе работы методов типа.

Реализация содержит внутренние детали и аспекты механизмов работы экземпляров типа. *Реализация* включает в себя *представление* и набор *методов*.

Представление – это структура данных, получаемая из определения абстрактного типа на ODL после отображения в конкретный язык программирования (*ODL to program language mapping*). Каждому атрибуту абстрактного типа соответствует переменная в его представлении.

Методы – это тела операций (процедур), объявленных в спецификации данного типа, написанные с использованием классов, предоставляемых ОО средой.

Тип может обладать несколькими реализациями, хотя обычно только одна из них используется в конкретной программе.

Разделение спецификации и реализации типа, при которой спецификация описывается независимым от языка программирования образом, обеспечивает прозрачный доступ к объекту в неоднородном компьютерном окружении (соответствующий интерфейс на стороне клиента генерируется специальной утилитой – ODL компилятором).

Зависимости между типами в объектной модели (ОМ) представляются двумя видами связей, которые отражают соответственно *наследование поведения* и *наследование состояния.*

2.2 Наследование поведения

Данному типу наследования соответствует понятие ***ISA-связи*** (*is–a–relationship*) – связи наследования между типами, выражающей обобщение, когда супертип наследует поведение подтипа. Объектная модель поддерживает множественные *ISA*-связи наследования между интерфейсами. На языке *ODL ISA*-связи представляются двоеточием:

```
interface Person{…};
interface Employee {...};
interface Professor : Employee {...}, Person {…};
interface Associate_Professor : Professor {...};
```

2.3 Наследование состояния

ODMG определяет также связь наследования (***extends***), когда наследуется состояние объекта класса:

```
class Person {
attribute string name;
attribute Date birthDate;
};
class EmployeePerson extends Person : Employee {
attribute Date hireDate;
```

```
attribute Currency payRate;
relationship Manager boss
inverse Manager::subordinates;
};
```

Extends-связь выражает наследование поведения и состояния типа и не допускает множественное наследование.

2.4 Агрегированные типы

Объектная модель *ODMG* поддерживает различные типы объектов-агрегатов, которые состоят из элементов литеральных или объектных (возможно, агрегированных) типов. В рассматриваемом подмножестве *OM* будут поддерживаться следующие агрегированные типы (t означает тип элементов агрегата):

```
Bag<t>
List<t>
Array<t>
```

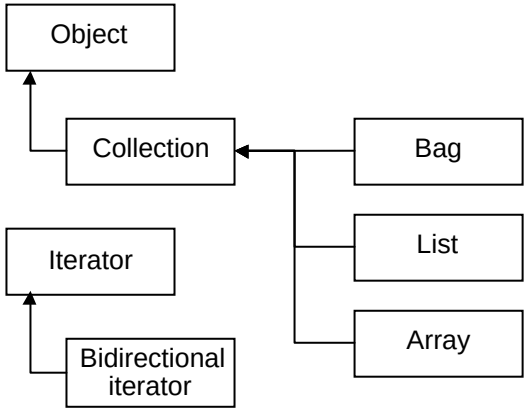

Рис. 2. Структура системы агрегированных типов.

Интерфейсы всех этих типов наследуются от типа Collection, который является абстрактным интерфейсом, объединяющим общие характеристики. Семантически Bag является *неупорядоченным* множеством с возможностью содержания одинаковых элементов. List это *упорядоченное* множество элементов. Array соответствует обычному понятию массива с возможностью динамического изменения размера. Ниже приведены описания этих агрегированных типов на ODL.


```
interface Collection : Object {
      exception          InvalidCollectionType;
      exception          ElementNotFound{any element;};
      unsigned long      cardinality ();
      boolean            is_empty();
      boolean            is_ordered();
      boolean            allows_duplicates();
      boolean            contains_element(in any element);
      void               insert_element(in any element);
      void               remove_element(in any element)
                         raises(ElementNotFound);
      Iterator           create_iterator(in boolean stable);
      Bidirectionallterator   create_bidirectionalIterator
      (in boolean stable) raises(invalidCollectionType);
};
```

Для доступа к элементам объекта `Collection` используется механизмы `Iterator` и `BidirectionalIterator`, которые определяются следующими интерфейсами:

```
interface Iterator {
      exception   NoMoreElementsO;
      exception   lnvalidCollectionType{};
      boolean     is_stable();
      boolean     at_end();
      void        reset();
      any         get_element() raises(NoMoreElements);
      void        next_position() raises(NoMoreElements);
      void        replace_element(in any element)
                  raises(lnvalidCollectionType);
};
interface Bidirectionallterator : Iterator {
      boolean     at_beginning();
      void        previous_position() raises(NoMoreElements);
};
```

От интерфейса Collection наследуются более детализированные агрегированные типы:

```
interface Bag : Collection {
      unsigned long      occurrences_of(in any element);
      Bag                create_union(in Bag other_bag);
      Bag                create_intersection(in Bag other_bag);
      Bag                create_difference(in Bag other_bag);
};

interface List : Collection {
      exception lnvalidlndex{unsigned long index};
```

```
      void  remove_element_at(in unsigned long index)
                  raises(lnvalidlndex);
      any         retrieve_element_at(in unsigned long index)
                  raises(lnvalidlndex);
      void  replace_element_at(in any element,
                  in unsigned long index)
                  raises(lnvalidlndex);
      void  insert_element_aftef(in any element,
                  in unsigned long index)
                  raises(lnvalidlndex);
      void  insert_element_before(in any element,
                  in unsigned long index)
                  raises(lnvalidlndex);
      void  insert_element_first (in any element);
      void  insert_element_last (in any element);
      void  remove_first_element()
                  raises(ElementNotFound);
      void  remove_last_element()
                  raises(ElementNotFound);
      any         retrieve_first_element()
                  raises(ElementNotFound);
      any         retrieveLast_element()
                  raises(ElementNotFound);
      List  concat(in List otherList);
      void  append(in List otherList);};
interface Array: Collection {
      exception   lnvalidlndex{unsigned long index};
      void        replace_element_at(in unsigned long index,
                  in any element)
                  raises(lnvalidlndex);
      void        remove_element_at(in unsigned long index)
                  raises(lnvalidlndex);
      any         retrieve_element_at(in unsigned long index)
                  raises(lnvalidlndex);
      void        resize(in unsigned long new_size);};
```

2.5 Атрибуты

Как уже упоминалось выше, состояние объекта определяется набором значений его *свойств,* подразделяющихся на *атрибуты* и *связи*. Атрибуты составляют внутреннее состояние объекта. Они могут быть элементарных и агрегированных литеральных или объектных типов. При этом последние выделяются особо, так как включение одним объектом в свой состав других объектов является типичным для ОО программирования. Они называются *атрибутами-ссылками.* Реально объекты ссылаются друг на друга посредством своих идентификаторов (объектной ссылки OID или идентификатора

хранимого объекта POID), поэтому значение атрибута-ссылки – это объектная ссылка OID некоторого объекта в приложении или POID в БД. При отображении хранимых объектов из БД в приложение идентификаторы хранимых объектов соответствующим образом преобразуются в объектные ссылки.

Заметим, что атрибуты являются *абстрактным* описанием состояния объекта и не обязаны соответствовать структурам данных при отображении в конкретный язык программирования. Они могут реализовываться и как набор соответствующих методов.

2.6 Связи

Тогда как *свойства-атрибуты* объекта характеризуют его внутреннее состояние, *свойства-связи* используются для задания семантических связей (отличных от связей наследования) с другими типами в спроектированной пользователем Объектной Модели Задачи (ОМЗ). Отметим, что *связи* – не то же самое, что атрибуты-ссылки. Атрибуты-ссылки характеризуют состояние объекта, которому принадлежат, путем задания ссылки на объекты, входящие в состав данного, и являются *однонаправленными* (отражают семантику *объекта-агрегата*), тогда как *связи* характеризуют не состояние объекта, а участие его в том или ином отношении к другим объектам в смысле семантических связей задачи и всегда являются *двунаправленными.* Хотя синтаксическая реализация этих свойств в конкретных языках программирования и совпадает.

Связи могут существовать между экземплярами только двух типов (которые могут совпадать), т. е. *связи* являются бинарными. Свойства-связи, также как и атрибуты, объявляются в *ODL*-схеме БД. В объявление связей входит обязательное указание на обратное направление связи. Ниже приведен пример связи двух классов `Professor` и `Course`.

```
interface Professor {
      relationship      set <Course> teaches
                        inverse Course:: is_taught_by;};
interface Course {
      relationship      Professor is_taught_by
                        inverse Professor:: teaches;
};
```

Реализуются связи, также как и атрибуты-ссылки, с помощью объектных ссылок (POID и OID) на связанный объект, массива ссылок, либо объектов-агрегатов таких ссылок. Связи по типу бывают *упорядоченные* и *неупорядоченные*, в зависимости от типа агрегата, представляющего связь. Например, bag<Course> – неупорядоченная связь, тогда как list<Member> – упорядоченная. Заметим, что упорядоченные связи практически мало применяются для моделирования реальных прикладных задач. Связи также различаются по *мощности* (количеству элементов с обеих сторон) и бывают следующих типов: 1:1, 1:*n*, *n:m*.

Механизм связей *ODMG* – *двунаправленный*. При этом СУБД обязана отслеживать корректность взаимных ссылок хранящихся в БД объектов согласно имеющейся схеме и обеспечивать целостность данных, в частности удаление связей на объект со стороны других объектов при его удалении из БД (обеспечение целостности для атрибутов-ссылок не гарантируется).

Описанные в формате *ODL* связи – лишь абстрактная спецификация. Для организации нормальной работы и использования связей необходимо описать некоторые классы и методы, позволяющие устанавливать, видоизменять и удалять эти связи. Так *ODL* – описание связи (1:1):

```
relationship X inverse Z;
```

преобразуется в эквивалентное *IDL* – описание:

```
attribute X, Y;
void form_Y(in X target);
void drop_Y(in X target);
```

Аналогично, для связей мощности 1:n имеем:

```
relationship set<X> Y inverse Z;
```

На *IDL*:

```
attribute    set<X> Y;
void   form_Y (in X target)
       raises(lntegrityError);
void         drop_Y (in X target);
void         add_Y (in X target)
       raises(lntegrityError);
void         remove_Y (in X target);
```

Соответствующие классы, обеспечивающие работу со связями во время выполнения, предоставляются ООС.

2.7 Исключения (exceptions)

Для обработки возникающих в процесс работы ошибок в *ODMG* применяется модель исключительных ситуаций, аналогичная используемой в C++ и Java. Исключения в ОМ являются объектами и обладают интерфейсом для связи их с другими исключениями и методами, которые могут вызывать или обрабатывать эти исключения.

2.8 Хранимые объекты(Persistent Objects)

Для приложения ООС является хранилищем перманентных (*хранимых) объектов*. При этом в конкретный момент времени такой объект может и не существовать в памяти программы. Но ООС знает уникальный идентификатор *(POID)* хранимого объекта и знает, как этот объект "синтезировать", так что по запросу пользователя (т.е. по вызову соответствующих методов ООС) он может

быть создан в памяти приложения и проинициализирован состоянием, хранимым в БД. Следует разделить понятия уникального Объектного Идентификатора в прикладной программе (***OID***) и уникального Объектного Идентификатора Хранимого Объекта в БД (***POID***). *OID* и *POID* представляют, вообще говоря, независимые пространства идентификаторов. При этом один и тот же хранимый объект, имеющий свой постоянный *POID* в БД, может обладать различными *OID* в прикладной программе при её запуске в разные моменты времени.

2.9 Сегмент (extent) БД

Сегментом объектного типа называется совокупность всех хранимых объектов данного типа в рамках определенной БД. Если тип *B* наследуется от типа *A*, то сегмент типа *B* является подмножеством сегмента типа *A*. Сегменты имеют большое значение для работы целевого приложения. Они являются той базовой единицей, по которой производятся *OQL* запросы на получение хранимых в БД объектов. Если провести параллель с реляционной БД, сегменту можно сопоставить реляционную таблицу.

2.10 Объекты с возможностью хранения

Все объекты приложения подразделяются на *объекты с возможностью хранения (persistent-capable, PC)* и *объекты без возможности хранения (non-persistent-capable, NPC)*, т. е. объекты, которые могут или не могут обладать свойством хранимости (*persistency*) соответственно. Реально свойство *PC* обеспечивается наследованием от *хранимых классов и интерфейсов*, предоставляемых ООС. Для того чтобы объект потенциально мог быть сохранен в БД, он должен быть *PC*, хотя это не означает, что всякий такой объект с необходимостью будет сохранен по окончании транзакции. В этом отношении пользователю предлагается гибкая схема, когда он может явно указать на необходимость сохранения конкретного *PC* объекта, либо обозначить все используемые в программе *PC* объекты как объекты, которые должны быть сохранены. Вследствие агрегирования одними объектами в своем составе других объектов (по объектным ссылкам), операция сохранения некоторого объекта может повлечь за собой необходимость сохранения объектов, на которые ссылается этот объект. В этой ситуации действует правило, что все те объекты, на которые ссылается исходный объект, также должны быть сохранены, и далее по иерархии. Это правило называется *хранимостью по достижимости* (*Persistency by Reachability (PBR)).*

2.11 Именованные хранимые объекты

Хранимому объекту можно присвоить одно или несколько *имен*. В приложениях наиболее часто практикуется такая схема работы, когда необходимо получить некоторый начальный объект, используя который, затем можно по объектным ссылкам получить доступ к другим нужным объектам. Имя и является той характеристикой, по которой можно получить начальный

объект из БД. Именованные объекты в БД называются *точками входа*. Имена являются уникальными в пределах конкретной БД. Именованный хранимый объект можно создать с помощью вызова метода `bind(Object object, String name)` класса `Database`, предоставляемого ООС:

```
public class Database {
      public static Database open
      (String name, int accessMode) throws ODMGException;
  public void close() throws ODMGException;
  public void bind();
  public Object lookup(String name)
              throws ObjectNameNotFoundException;
  public void unbind(String name)
              throws ObjectNameNotFoundException;
}
```

Кроме того, в классе `Database` определены методы для получения именованного объекта из БД – `lookup()`, отключения имени от объекта `unbind()` и собственно открытия `open()` и закрытия `close()` базы данных.

2.12 Создание объектов

Объекты с возможностью хранения создаются так же, как и обычные объекты приложения, т.е. с помощью вызова конструктора соответствующего класса. При этом по окончании формирования объекта в памяти программы, в БД этого объекта еще не существует. До выполнения первой операции сохранения нового объекта нет никакого объекта в БД, связанного с данным объектом приложения. Когда же была выполнена операция сохранения объекта в БД, ООС реально создает хранимый объект в БД, который в некотором смысле является образом объекта в приложении, и сохраняет в нем текущее состояние нашего объекта. Созданный в БД хранимый объект обладает собственным уникальным идентификатором POID.

2.13 Поддержка целостности БД. Множественные ссылки

В объектной модели задачи может реализоваться ситуация, когда на один и тот же объект могут ссылаться несколько других, включающих его в себя, объектов. По этой причине в некоторых операциях с БД (сохранить, считать из БД) могут возникать дублирующиеся операции над одними и теми же объектами при распространении начальных операций по объектным ссылкам (см. ниже описание схем выполнения операций сохранения, удаления объектов). В других операциях (удалить) могут возникать неразрешимые ссылки, когда один из ссылающихся объектов удаляет по ссылке объект из БД, после чего другой уже не может разрешить свою ссылку на удаленный объект. Для устранения этих несоответствий, с каждым объектом в БД и каждым объектом в приложении (для приложения обычно выполняется самой объектной средой) необходимо связать счетчик числа ссылок на этот объект со стороны других объектов. Тогда операция удаления объекта из БД будет уменьшать счетчик ссылок на объект до

тех пор, пока он не обнулится, и тогда объект реально будет удален из БД. Повторное считывание одного и того же объекта из БД предотвращается тем, что ООС будет использовать кэширующую схему, когда каждый хранимый объект, считываемый из БД, будет фиксироваться в специальном буфере системы, и повторные запросы на получение уже считанных объектов будут возвращать OID существующего в памяти объекта.

2.14 Получение хранимых объектов из БД

Ранее сохраненные в БД хранимые объекты могут быть получены из БД и отображены в объекты приложения соответствующего класса. Существует два способа получения хранимых объектов из БД:

- по имени объекта, используя метод `lookup()` класса Database.
- с помощью OQL-запроса к БД.

В первом случае метод `lookup()` создает в памяти приложения объект того класса, которому принадлежит соответствующий именованный хранимый объект в БД и инициализирует его состоянием этого хранимого объекта. Метод `lookup()` возвращает OID вновь созданного в памяти объекта.

Во втором случае создается объект-агрегат, который включает ссылки на множество возвращаемых объектов, которые создаются *OQL*-запросом в памяти приложения. Описание реализуемого подмножества *OQL* приводится ниже.

Более гибко выглядит схема создания объекта-агрегата, содержащего неразрешенные ссылки на целевые объекты запроса, которые разрешаются по мере того, как пользователь обращается по этим ссылкам. При попытке пользователя обратиться к какому-то объекту по объектной ссылке ООС времени исполнения “подкачивает этот объект” из БД, т. е. реально создает объект в памяти приложения, инициализирует его состоянием соответствующего хранимого объекта и устанавливает значение ссылки в объекте-агрегате, равное ссылке на созданный в памяти объект. Такой подход позволил бы в некоторых случаях сэкономить системные ресурсы, когда не все из запрошенных объектов будут использованы в программе, а также сэкономить время обработки OQL-запроса, асинхронно возвращая управление программе пользователя. С другой стороны, при таком подходе дополнительное время будет тратиться в процессе работы на “подкачку” очередного объекта в программу. Для получения очередного объекта из объекта-агрегата будет использоваться класс-итератор, который и будет активизировать методы, осуществляющие подкачку очередного объекта в память приложения.

2.15 Сохранение PC объектов в БД

Для сохранения некоторого объекта с возможностью хранения в БД, используется хранимый метод этого объекта save(), содержащийся в унаследованном хранимом классе ООС. Операция сохранения *PC* объекта

выглядит единообразно для всех объектов, независимо от того способа, которым он был создан в приложении. Объект мог быть создан ООС с образа ранее существовавшего в БД хранимого объекта (например, с помощью OQL запроса), и тогда метод `save()` произведет обновление состояния соответствующего хранимого объекта в БД текущим состоянием сохраняемого объекта. Иначе, объект мог быть сконструирован конструктором соответствующего класса, вызванного самой прикладной программой, и тогда метод `save()` активизирует алгоритм создания нового хранимого объекта в БД и сохранит в нем состояние нашего объекта.

2.16 Удаление хранимых объектов из БД

Для удаления некоторых хранимых объектов из БД, необходимо просто указать ООС, какие именно хранимые объекты (по какому признаку) должны быть удалены. Простейшим способом можно удалить хранимый объект, когда в программе существует соответствующий ему образ, ранее созданный из данного хранимого объекта. Между объектами с возможностью хранения приложения и хранимыми объектами БД в каждый момент времени существует взаимно однозначное соответствие (устанавливаемое между *OID* и *POID*). Поэтому, для выполнения операции удаления объекта вызывается метод `delete (Object o)` класса `Database`, которому передается объект в памяти приложения `(Object o)`. Все объекты, входящие в состав удаляемого объекта , также должны быть удалены, что осуществляется рекурсивно. Все семантические связи с объектами ОМЗ также должны быть удалены с обеих сторон, что выполняется ООС. После удаления из БД хранимого объекта, в памяти еще остается соответствующий ему объект, который как бы “теряет” свой образ в БД, но с которым приложение может продолжать работать. Впоследствии этот объект может опять быть сохранен в БД, и для него будет создан новый хранимый объект (с новым POID), с которым он будет ассоциирован. Класс Database содержит также методы `delete(<collection> c),` которым могут передаваться агрегаты объектов для удаления.

2.17 Пользовательские сессии, транзакции. Сервисы

Объектно-ориентированная СУБД, как и всякая другая, поддерживает концепции пользователей, прав доступа к объектам БД, пользовательских сессий и других понятий, обеспечивающих возможности по разделению БД, как общего ресурса. Все вышеперечисленные и прочие возможности могут реализовываться как отдельные сервисы ООС, реализуемые в библиотеках классов, которые предоставляются прикладному программисту для использования в процессе написания программ. Эти сервисы могут быть надстроены на базе основной (*core*) объектной модели. В первой версии системы данные сервисы не будут реализованы. Они могут быть добавлены на дальнейших этапах разработки. Таким образом, будет реализовываться

однопользовательская система хранения объектов, где клиентское и серверное приложения работают на одной и той же машине. Концепция транзакций в базовой версии будет реализована в объеме, специфицированном стандартом *ODMG*. Методы класса, реализующего транзакции, будут активизировать операции по обмену данными с *СДХ*, например, вызов метода commit() запустит алгоритм сохранения состояний изменившихся объектов в БД.

3. Язык определения объектов (ODL)

ODL – язык, используемый для определения спецификаций объектных типов в соответствии с объектной моделью *ODMG*. Он предназначен для обеспечения переносимости схем объектных баз данных между различными объектными БД и является аналогом языка определения данных (*DDL*) в реляционных БД. *ODL* предполагает определения объектов, которые могут быть реализованы в различных языках программирования, и поэтому не зависит ни от какого конкретного языка. В настоящее время поддерживается преобразование (меппинг) конструкций *ODL* в *C++, Smalltalk* и *Java*. В данном разделе рассматривается базовое подмножество *ODL* и преобразование типов ОМ ODMG в язык *Java*. Полное описание ODL можно найти в [1].

3.1 Спецификация ODL

3.1.1 Определение структуры файла ODL

Определения *ODL* делятся в нашем случае на описания исключений, интерфейсов и классов.

```
<specification>   ::= <definition>
                        | <definition><specification>
<definition>            ::= <except_dcl>;| <interface>;
                        | <class>;
<interface>       ::= <interface_dcl>
                        | <forward_dcl>
<interface_dcl>   ::= <interface_header>
                         {[<interface_body>]}
<forward_dcl>           ::= interface <identifier>
<interface_header>      ::= interface <identifier>
                        [<inheritance_spec>]
<class>                 ::= <class_header> {<interface_body>}
<class_header>          ::= class <identifier>
                        [extends <scopedName>]
                        [<inheritance_spec>]
                        [<type_property_list>]
```

Таким образом, файл описаний *ODL* состоит из набора определений этих объектных типов, которые могут чередоваться в любом порядке. Рассмотрим подробней их характеристики.

```
<inheritance_spec>      ::=
:<scoped_name>[,<inheritance_spec>]
<type_property_list>::= ( [extent_spec][<key_spec>] )
<extent_spec>           ::= extent <string>
<key_spec>        ::= key[s] <key_list>
<key_list>        ::= <key> | <key>, <key_list>
<key>             ::= <property_name>|(<property_list>)
<property_list>   ::= <property_name>
                      | <property_name>, <property_list>
<property_name>   ::= <identifier>
<scoped_name>           ::= < identifier>| :: < identifier>
                      | <scoped_name> :: <identifier>
```

В каждом определении класса может быть не более одного описания сегмента и ключа, но при этом самих ключей может быть более одного. Наследование, определяемое **extends**, задает связь типа extends, а интерфейсы, указываемые после двоеточия, задают связи типа IS-A. Приведем несколько примеров, использующих указанные возможности:

```
interface    Student-IF
{interface_body    }

class  Postgraduate extends    Student: Student-IF
(
 extent      students
 key   ( name, group)
)
{      interface_body    }
```

3.1.2 Спецификация тела интерфейса

В теле интерфейса или класса определяются атрибуты и операции экземпляров данных типов. Атрибуты делятся на атрибуты-данные и атрибуты-связи. Связи всегда двунаправлены. Если связь указана в одном определении, то она должна быть описана и в определении типа, на который ссылается источник. Описания операций полностью соответствует спецификации *OMG IDL. BNF* спецификация определяется следующим образом:


```
<interface_body>  ::= <export>
                      | <export><interface_body>
<export>              ::=   <except_dcl>;| <attr_dcl>;
                      | <rel_dcl>;| <op_dcl>;

<attr_dcl>        ::= attribute <domain_type><attribute_name>
                      [ <fixed_array_size> ]
<domain_type>     ::=   <simple_type_spec>| <struct_type>
                      | <enum_type>
<rel_dcl>             ::=   relationship
                      <target_of_path><identifier>
                      inverse <inverse_traversal_path>
<target_of_type>  ::=   <identifier>
                      | <rel_collection_type>< <identifier> >
<inverse_traversal_path>::=   <identifier> :: <identifier>
<op_dcl>              ::=<op_type_spec>
                      <identifier><parameter_dcls>
                      [ <raises_expr> ]
<op_type_spec>        ::=   <simple_type_spec>
                      | void
<parameter_dcls>  ::=   ( [<param_dcl_list>] )
<param_dcl_list>  ::=   <param_dcl>
                      | <param_dcl>, <param_dcl_list>
<param_dcl>       ::=<param_attribute><simple_type_spec>
                      <declarator>
<param_attribute> ::=   in | out | inout
<raises_expr>         ::=   raises ( <scoped_name_list> )
<scoped_name_list>    ::=   <scoped_name>
                      | <scoped_name>, <scoped_name_list>
```

3.2 Пример описания ODL

Приведем описание схемы институтской иерархии, которое соответствует графическому представлению, изображенному на рис. 3.

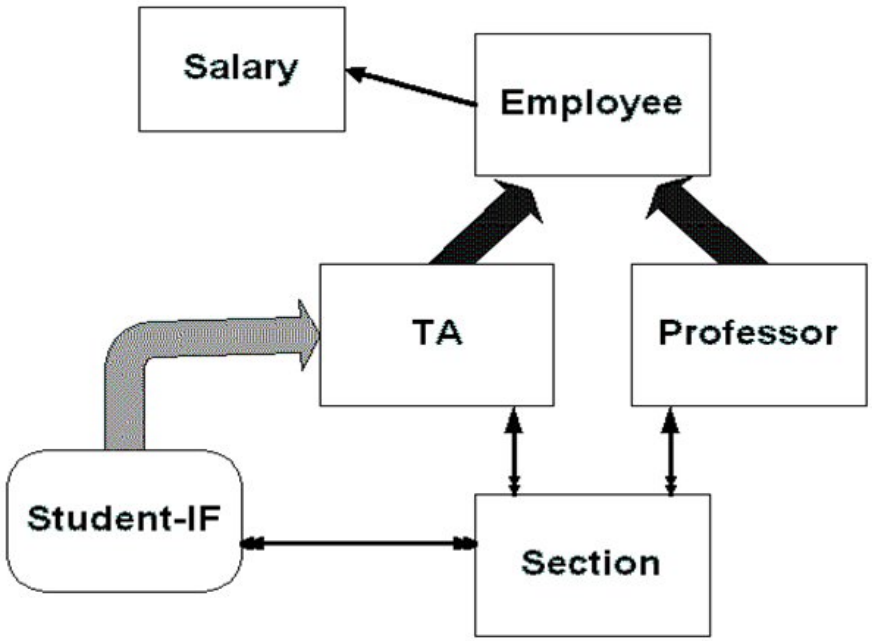

Рис. 3. Графическое представление схемы ODL.

```
// Класс Зарплата
class       Salary
{
      attribute   float base;         // оклад
      attribute   float overtime;           // сверхурочные
      attribute   float bonus;              // премия};
// Класс Служащий
class Employee
(
      extent      employees                 // сегмент Служащие
      key         id                        // ключ ID
)
{
      attribute   string  name;             // ФИО
      attribute   short   id;               // идентификатор
      attribute   Salary  annual_salary;    // зарплата
      void        hire();                   // нанять служащего
                                            // уволить служащего
      void        fire() raises ( no_such_employee );
```



```
};
// Класс Профессор наследует состояние Служащего
class Professor extends Employee
(     extent  professors      )     // сегмент Профессоры
{
// связь с Секциями (профессор преподает несколько секций)
      relationship   set<Section> teaches
            inverse     Section::is_taught_by;// обратная связь
      short grant_tenure()                    // избрать
      // возможное исключение
                  raises (ineligible_for_tenure);
};
// интерфейс Студент
interface Student-IF
{     attribute   string  name;               // ФИО
      attribute   string  student_id;         // идентификатор
      // связь с секциями (студент изучает несколько секций)
      relationship  set<Section>    takes
            Inverse     Section::is_taken_by;
};
// Класс Ассистент наследует состояние Служащего и поведение Студента.
class TA    extends Employee : Student-IF
{
      // связь с секцией ассистента
      Relationship  Section assist
            inverse     Section::has_TA;
      attribute   string  name;               // ФИО
      attribute   string  student_id;         // идентификатор
      // связь с секциями (ассистент изучает несколько секций)
      relationship  set<Section>    takes
            inverse     Section::is_taken_by;
};
```

3.3 Соответствие типов данных ODL и Java

Файл *ODL*, создаваемый программистом для описания схемы классов приложения в независимой от языка программирования форме, подлежит в конечном итоге трансляции в конструкции конкретного языка. В рамках нашей работы нас интересует преобразование типов из *ODL* в *Java*, которое складывается из стандартов ***ODMG 2.0*** и ***OMG CORBA 2.2.*** Соответствие между основными конструкциями ODL и Java представлено в таблице.

Приведем несколько примеров (названия классов и элементов абстрактные), иллюстрирующих как будет преобразовано описание схемы классов на *ODL* в конструкции языка *Java:*

ODL	Java
Struct	Class
Set	public interface Set extends Collection { … }
Bag	public interface Bag extends Collection { … }
List	public interface List extends Collection { … }
Iterator	Java Enumeration interface
Array	Primitive array type[] or Java Vector class
Relationship	Атрибут класса*
Extent	public interface Extent extends Collection { … }*
Exception	Java exception
Long	int, Integer (class)
Short	short, Short (class)
Unsigned long	long, Long (class)
Unsigned short	int, Integer (class)
Float	float, Float (class)
Double	double, Double (class)
Boolean	boolean, Boolean (class)
Octet	byte, Integer (class)
Char	char, Character (class)
String	String

* не поддерживается стандартом

4. Объектный язык запросов (OQL)

Группой *ODMG* разработан стандарт языка запросов для объектно-ориентированных баз данных ***Object Query Language (OQL).*** Это декларативный язык запросов, базирующийся на языке запросов реляционных баз данных *SQL*. *OQL* предназначен для получения объектов, уже хранящихся в БД. Основная конструкция *OQL* выглядит так же, как и в *SQL*. Это выражение вида:

```
SELECT <target> FROM <extents> WHERE <condition>
AGREGATE_SORT_CRITERIA
```

OQL разрабатывался как объектно-ориентированный язык. Результатом запроса является набор объектов (объект-агрегат, содержащий объекты целевого типа). По сложным объектным связям между объектами, атрибутам и методам можно производить навигацию, используя стандартный объектно-ориентированный синтаксис – оператор “.”. *OQL*-запросы в программе тесно

связаны со схемой БД, созданной в *ODL*-описании на этапе разработки БД. Имена, фигурирующие в частях `<target>`, `<extent>`, `<condition>`, и т. д. – это имена, объявленные в *ODL*-описании БД. Рассмотрим части *OQL*-запроса.

<target> – имя типа целевого объекта (набора объектов) или итерационной переменной, объявленной в части `<extents>`.

<extents> – сегменты объектов, из которых производится выборка. Это могут быть сегменты, объявленные в ODL-схеме, либо производные сегменты. В этой части запроса могут использоваться итерационные переменные. Ниже на примерах будут продемонстрированы оба случая.

<condition> – критерий на составляющие части целевого типа, согласно которому производится выборка объектов.

AGREGATE_SORT_CRITERIA – часть запроса, содержащая критерии агрегирования **(**`GROUP BY`**)**, сортировки **(**`HAVING`**)** и упорядочения результатов запроса **(**`ORDER BY`**)**. В нашей работе мы будем реализовывать только часть `ORDER BY`, имеющую следующий синтаксис:

```
ORDER BY <members>
```

<members> – критерий на составляющие части целевого типа, согласно которому производится упорядочение полученного набора объектов. Таким образом, общий вид OQL запроса в рассматриваемой конфигурации:

```
SELECT <target> FROM <extents> WHERE <condition>
ORDER BY <members>
```

Результатом запроса может быть упорядоченный или неупорядоченный список объектов (`<Bag>` и `<List>` соответственно), в котором может содержаться один объект, либо `NULL`-объект, когда запрос не возвращает объектов.

Теперь рассмотрим подробно возможные виды частей запроса и приведем примеры OQL-запросов, демонстрирующие возможности OQL.

В простейшем случае поле `<extents>` содержит имя сегмента ODL-схемы, а поле `<target>` содержит имя соответствующего этому сегменту типа в ODL-схеме или имя итерационной переменной, соответствующей этому типу. Напомним, что понятие типа объединяет в себе класс (class) и интерфейс (interface).

```
select p from Persons p;
```

Данный запрос вернет набор всех объектов, принадлежащих сегменту `Persons` класса `Person`, объявленному в *ODL*-схеме. В запросе используется итеративная переменная “p”.

В следующем примере демонстрируется случай, когда в поле `<extents>` фигурирует производный сегмент, содержащий все объекты ”всех объектов”, хранящихся в БД. Заметим, что в этом примере в поле `<extents>` используется навигация по объектной связи `Person -> Child (Children)`:

```
select c from Persons p, p.children c
```

В поле <condition> в простейшем случае фигурирует выражение над атрибутами целевого типа:

```
select p from Persons p where p.name = “Jane”
```

Подобно языку *SQL* можно сформировать запрос с соединением нескольких сегментов (join):

```
select p from Persons p, Flowers f where p.name=f.name
```

Теперь приведем пример запроса с навигацией по объектным связям. В простом случае связи типа 1:1:

```
select p.spouse.address from Persons p
```

В этом примере целевым объектом являются объекты-адреса супругов всех объектов типа “person”, хранящихся в БД. Теперь приведем пример запроса с навигацией по связи типа n:1. Допустим, мы хотим получить объекты-адреса детей всех объектов”, хранимых в БД:

```
select c.address from Persons p, p.children c
```

В этом случае мы фактически строим производный сегмент всех объектов ”всех объектов”, по которому затем производим выборку. Наконец рассмотрим навигацию в запросе типа n:m. Допустим, в нашей объектной модели описаны группы сотрудников такие, что в каждой из них может содержаться много сотрудников и в то же время каждый сотрудник может принадлежать нескольким группам одновременно. Тогда запрос для получения адресов всех сотрудников, принадлежащих всем группам, занимающимся программированием (таких групп много), выглядит так же, как и предыдущий запрос:

```
select m.address from Groups g, g.members m where
g.topic = “programming”
```

Однако в этом случае один и тот же объект типа “member” может принадлежать двум или более группам с темой “programming”. Но поскольку реально это один и тот же объект, а каждый объект обладает своим уникальным идентификатором в БД, дублирования в результирующем наборе объектов, который вернет запрос, не будет. В этом смысле объектный запрос всегда работает как “distinct” в SQL.

Объектно-ориентированная парадигма распространяется и дальше на язык OQL, и в запросах можно с помощью оператора “.” не только производить навигацию по объектным связям, обращаться к атрибутам объектов, но и вызывать методы объектов. При этом вызываемый метод обязан вернуть значение литерального типа, либо объект. Когда метод возвращает литерал,

навигация к методу по типу возвращаемого результата ничем не отличается от навигации к атрибуту, и может использоваться в части <condition> и <members> запроса. Когда метод возвращает объект, навигация к методу по типу возвращаемого результата не отличается от навигации по объектной связи, и может использоваться в частях <target> и <extents> запроса. Ниже приводятся примеры

```
select x from Students x
where x.count_lectures() < 10
```

В этом примере производится навигация к методу count_lectures() объекта типа “student”, возвращающему численный литерал, который затем используется в критерии запроса.

```
select x from Students x order by x.count_lectures(), x.name
```

В этом примере метод count_lectures() возвращает численный литерал, по которому производится дальнейшее упорядочение результатов запроса.

```
select p.activities().leader from Persons p
where p.age < 10
```

Здесь метод activities класса “person” возвращает объект класса “activity”, а затем производится навигация к объекту класса “person”, связанного с классом “activity”.

```
select a from Persons p, p.activities() a
```

В этом примере все аналогично, только навигация к методу, возвращающему объект, находится в части <extents> запроса, за счет чего создается новый сегмент.

Часть <condition>, содержащая критерий выборки, в общем случае является объединением нескольких boolean выражений по типу AND или OR. В BNF-нотации реализуемого подмножества OQL приводятся все операторы, которые могут быть использованы в критерии. Также допустимы метасимволы в выражениях критерия выборки.

В части <members>, содержащей критерий упорядочения, перечисляются через запятую атрибуты целевого типа или методы, возвращающие литерал, по которым нужно упорядочить результат запроса с указанием направления сортировки (ASC – в порядке возрастания, DESC – в порядке убывания).

OQL будет реализован как встроенный язык запросов. В OQL-выражении могут использоваться свободные переменные, которые входят в состав абстрактного шаблона OQL-запроса. Перед выполнением такого запроса необходимо связать все свободные переменные с реальными переменными. Для выполнения запросов в программе предусмотрен специальный класс OQLQuery, используемый для формирования запросов, связывания свободных переменных и собственно выполнения запроса:

```
class OQLQuery {
  public OQLQuery() {}
  public OQLQuery(String query) {…}
  public create(String query) {… }
  public bind(Object parameter) {… }
  public Object execute() throws ODMGException {…}
}
```

В подмножестве OQL мы не будем реализовывать вложенные запросы.

5. Разработка конечного приложения с использованием системы

Опишем общую схему разработки приложений в рамках рассматриваемой системы (см. рис. 4). Отметим действия, предпринимаемые программистом при разработке приложения, и опишем информационные потоки, возникающие в процессе эксплуатации системы.

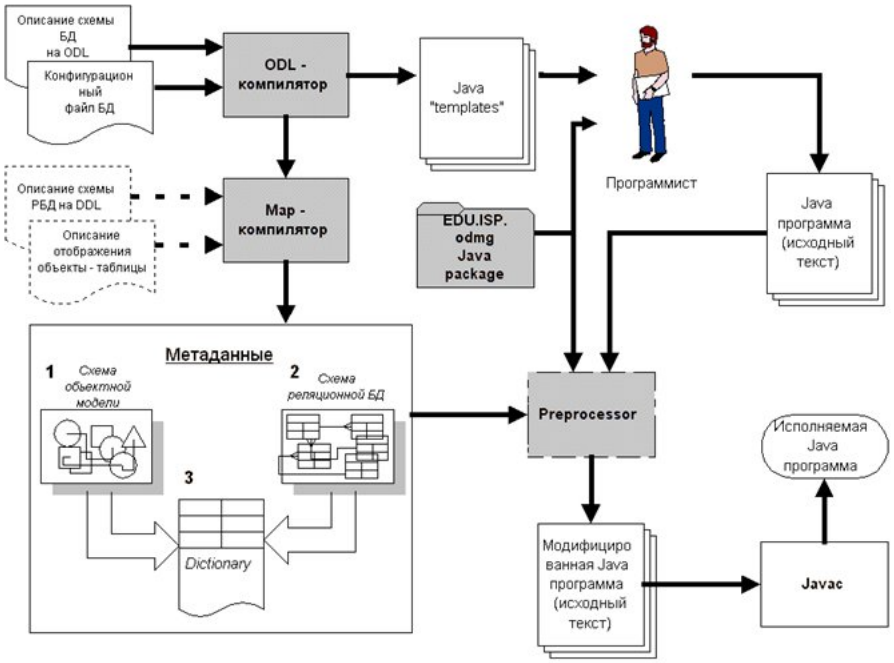

Рис. 4. Компоненты ОО среды и их взаимодействие.

В первую очередь рассматриваем задачу *прямого проектирования*. На начальном этапе разработки программист создает схему, описывающую классы и связи между ними, используя ODL. При этом в каждом классе определяются атрибуты

и методы, задаются исключения. Необходимо также создать еще файл конфигурации, в котором описывается определенная служебная информация, в частности типы классов (persistent/transient), способ отметки модификации объектов определенного класса (manual/auto) и т. п.

Далее это описание направляется на вход компилятору ODL, который создает заголовки соответствующих классов на Java. На данном этапе у программиста имеются “шаблоны” интерфейсов и классов его приложения. В описаниях отражены все атрибуты (с типами, соответствующими языковому отображению (language binding)), все операции и исключения, сформированы скелеты конструкторов. То есть делается все, чтобы максимально облегчить механическую работу по набиванию текста. Не хватает только кода, реализующего соответствующие методы и задающего собственно семантику работы (сохранение целостности данных становится задачей ООБД). С точки зрения работы системы данный этап является основным по накоплению начальной информации. В системных файлах создаются описания объектной схемы (1), файлы служебной информации, генерируется реляционное представление объектов (2), а также информация по мэппингу между реляционным и объектным представлением – Dictionary (3), т. е. создаются так называемые метаданные.

Теперь программист может использовать сгенерированные описания в своей среде программирования Java и писать приложения привычным для себя образом, используя возможности ООБД путем вызова методов объектов, предоставленных в package RU.ISPRAS.odmg *(название предварительное)*.

По завершению определенного этапа разработки, созданные файлы передаются на вход *ODMG preprocessor*, где к созданным программистом описаниям классов и функций добавляются описания и действия, реализующие собственно работу ООБД. Хотя из-за ограничений на использование препроцессоров в Java, возможен перенос данной нагрузки на ODLCompiler и собственно программиста.

Все классы с возможностью хранения модифицируются для наследования интерфейса ODMGObject (название предварительное), в места, где используются или модифицируются хранимые объекты, вставляются вызовы служебных методов fetch() и durty(). Система проверяет правильность использования механизмов *долговременного хранения*. ODMG preprocessor использует во время своей работы информацию из системного Dictionary и может выдавать сообщения об ошибках. В результате получается набор Java файлов, которые автоматически передаются компилятору javac, который генерирует собственно результат работы – набор class файлов. Далее их можно использовать обычным способом, передавая на Java интерпретатор.

Процесс *обратного проектирования* на этапе работы не отличается от вышеизложенного. Отличие заключается в способе задания соответствия между реляционной и объектной моделями (Dictionary). Если при *прямом проектировании* представление объектов в *РСУБД* определяется системой и

генерируется на основании ODL описания, то при обратном проектировании это представление уже задано, и нам нужно создать Dictionary на основании специально предоставляемой пользователем информации. А именно, программист сам задает схему РБД и меппинг из объектной модели в реляционную путем описания файлов DDL и MAP. Эти файлы используются программой MapCompiler для преобразования в стандартный формат метаданных. Таким образом, после обработки исходных данных ODLCompiler'ом и MapCompiler'ом, получается тот же набор информации, который используется унифицированным способом.

Итак, в архитектуре системы можно выделить следующие компоненты, подлежащие реализации:

```
ODLCompiler (executable)
MapCompiler (executable)
Preprocessor (executable)
JavaAPI package (RU.ISPRAS.odmg)
```

Во время выполнения программы необходимо обеспечить доступ к системным файлам (метаданным), созданным ODLcompiler, так как они активно используются при выполнении функций по записи/чтению из БД. Каждый запрос клиента разбирается Посредником с использованием информации, хранящейся в Dictionary, преобразуется в соответствующий SQL-запрос и поступает в буфер запросов к серверу *РСУБД*. В Dictionary хранится вся необходимая информация о том, в каких таблицах *РСУБД* хранятся запрошенные объекты (детали объектов). *РСУБД* возвращает результат запроса в буфер результатов SQL-запросов Посредника. Получая эту информацию, Посредник начинает строить множество затребованных в OQL-запросе объектов, выделяя под них память и инициализируя объекты информацией, полученной в результате выполнения соответствующего SQL-запроса сервером *РСУБД*. Сформированные объекты передаются приложению, которое может манипулировать ими как всеми прочими объектами, созданными конструкторами самого приложения, а также применять к этим объектам методы, выполняющие фиксацию изменений в БД, удаление объектов, и т. д. Эти методы активизируют соответствующие действия Посредника по сохранению в *РСУБД* изменений, удалению информации, и т. д.

6. Заключение

В статье был рассмотрен «верхний» компонент Посредника – *Объектно-ориентированная среда*. Эта среда связывает приложение с «нижним» уровнем системы – Сервисом долговременного хранения (СДХ). При этом, интерфейсы, предоставляемые приложению, соответствуют стандарту на ООБД ODMG 2.0. Была подробно изложена соответствующая объектная модель, введены основные понятия и термины, объяснены механизмы работы. В качестве средств доступа к хранимым объектам представлены язык запросов OQL и соответствующие классы. Также специфицировано подмножество языка определения объектов

ODL и описано отображение конструкций данного языка в Java. Тем самым мы определили все те компоненты (классы и утилиты), с которыми сталкивается прикладной программист в процессе создания приложения, использующего возможности ООБД. И, наконец, в последней главе описана комплексная схема создания приложений, выделены бинарные утилиты (ODLCompiler, MapCompiler, Precompiler) и специфицирован порядок их применения. При этом рассмотрены задачи как *прямого*, так и *обратного проектирования*. "Нижний" компонент посредника, как уже отмечалось выше, представлен JDBC интерфейсами, реализуемыми стандартными драйверами. Описание этих интерфейсов можно найти в [13].

Литература

1. R.G.G. Cattell, Douglas K. Barry. "The Object Database Standard: ODMG 2.0", 1997
2. R.G.G. Cattell. "The Object Database Standard: ODMG-93", 1996
3. "The Common Object Request Broker: Architecture and Specification", Revision 2.0, July 1995
4. "Persistent Object Service specification", OMG Document 94-10-7, 1994
5. "ObjectStore API User Guide", "ObjectStore API Advanced User Guide" Release 5.0 ObjectDesign Mar. 1997 http://wwwdb.informatik.uni-rostock.de/~jo/ostore5/user1/8_schema1.html
6. "The POET Java ODMG Binding", http://www.poet.com/
7. "POET Java SDK Programmers Guide", http://www.poet.com/
8. David Tothwell "Java Object Persistence", Technical Overview, Sep. 1996
9. Jan Klendienst, Frantisek Plasil, Petr Juma "Lessons Learned from Implementing the CORBA Persistent Object Service", Charles University
10. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen "Object-Oriented Modeling and Design" Prentice Hall, 1991
11. G. Booch "Object-Oriented Design" 1992, Behjamin/Cummings Publishing Company
12. Loomis Mary E. S. "Object Versioning" Journal of OO programming 1992
13. "Java® 2 Platform API Specification", Sun Microsystems, http://java.sun.com/products/jdk/1.2/docs/api/index.html