УДК 681.3

СРАВНИТЕЛЬНА ХАРАКТЕРИСТИКА МЕТОДИК ОБЪЕКТНО-РЕЛЯЦОННОГО ПРЕОБРАЗОВАНИЯ

И.А. Лихацкий

Институт программных систем НАН Украины
03680, Киев, проспект Академика Глушкова,40, корп. 5.
E-mail: igor_md@ukr.net

Описана методика создания объектно-реляционного преобразования с помощью системы C-Gen кодогенерации для автоматического создания уровня сохраняемости основанного на структуре базы данных. Использование текстовых шаблонов времени выполнения позволяет сгенерировать SQL код и API уровня сохраняемости. Проведена сравнительная характеристика системы кодогенерации C-Gen и Entity Framework, а также показана высокая эффективность предложенного подхода.

We describe a method to create object-relational transformation components by using code generation system to automatically generate persistence layer based on the database structure. Text template engine is used to generate SQL queries, business classes and APIs to access data from application code. There was done a comparison of the C-Gen system and Entity Framework and demonstrated high efficiency of proposed approach.

Введение

В настоящее время большинство бизнес-приложений создаются с помощью объектно-ориентированных языков программирования, таких как Java, C#, C++, а хранения данных осуществляется в реляционных базах данных, таких как Oracle, Microsoft SQL Server или MySQL. Как объектно-ориентированная так и реляционная парадигмы предоставляют надежный фундамент для решения прикладных задач. Они поддерживаются многочисленными инструментами и методологиями, которые позволяют упростить разработку и улучшить качество программного кода.

Тем не менее, модели данных для отображения объектно-ориентированных и реляционных систем существенно различаются. Поэтому разработчики сталкиваются с необходимостью преобразования данных из объектно-ориентированной формы в реляционную. Иногда такое преобразование осуществляется вручную, и в этом случае можно получить более эффективный программный код. Для реализации данного подхода разработчики должны выполнить весьма значительную часть рутинной работы, вручную осуществляя маппинг каждого объекта базы данных в их объектное представление. В другом случае использование автоматизированных инструментов (например, ORM систем [1]) может в значительной степени повысить производительность труда разработчиков, но зачастую это происходит в ущерб производительности приложения. Существует необходимость в нахождения такого решения, которое сочетало бы в себе производительность подхода ручного кодирования и автоматизацию присущую ORM системам.

Актуальность нахождения такого решения сочетается с известными преимуществами реляционных баз данных, таких как:

- возможность поддержки наследуемых (legacy) систем, использующих традиционные решения;
- простота использования технологии, основанной на понятной табличной модели и математически строгой теории реляционной алгебры;
- широкое распространение и тщательная многолетняя апробация предлагаемых на рынке продуктов СУБД;
- предоставление естественных для приложения объектно-ориентированных интерфейсов реализованных для популярных языков программирования.

В данной статье мы рассмотрим предложенный подход формирования объектного представления реляционных данных, целью которой является достижение высокого уровня производительности и значительного уровня автоматизации. В качестве решения данной проблемы предлагается создать объектную форму представления реляционной СУБД, объекты которой отражают сущности предметной области и однозначно связаны с таблицами, представлениями и хранимыми процедурами СУБД. Для автоматизации процесса генерации программного кода использовать текстовые шаблоны T4 времени выполнения [2], которые могут быть использованы как для генерации SQL кода, так и объектно-ориентированных интерфейсов для доступа и управления хранимыми данными. Существенно, что значительная часть функций по реализации указанных свойств ложится на промежуточный слой (persistent layer), а его создание сопряжено с комплексом проектных решений затрагивающих сопровождаемость, производительность, простоту применения клиентским приложением.

Объектно-ориентированная и реляционные парадигмы

Согласно парадигме объектно-ориентированного программирования, приложения обрабатывают данные которые представлены в памяти в виде графа объектов (коллекции взаимосвязанных объектов). Каждый объект имеет свойства которые могут быть представлены в виде атомарных типов данных либо в виде ссылок на дру-

гие объекты. Однако объектно-ориентированная парадигма, по умолчанию, не представляет средств для сохранения состояния объектного графа в файле или базе данных [3].

В качестве решения проблемы можно использовать реляционные базы данных для сохранения состояния объектной модели. В месте с тем, реляционные базы данных используют другой подход к организации и хранению данных. Сущности сохраняются в виде набора таблиц и связей между ними. Простые свойства хранятся в виде полей таблицы, в то время как ссылки хранятся в виде отдельных таблиц связанных между собой реляционным отношением. Таблицы могут быть связаны отношениями один-к-одному, один-ко-многим, многие-ко-многим. Поэтому один объект в ООП может быть представлен набором связанных таблиц в реляционном виде. Нарду с этим, в реляционных базах данных не поддерживается основные принципы ООП, такие как инкапсуляция, наследование, полиморфизм. Уникальность записей в таблице определяется на основе первичного ключа, в то время как уникальность объекта определяется областью памяти в которой он хранится.

Объектная и реляционные модели значительно отличаются по своей структуре. Это несоответствие называется **несоответствием парадигмы** (paradigm mismatch [4]), таким образом существует необходимость в компонентах которые осуществляли бы преобразование данных из объектного вида в реляционный и наоборот.

Одним из подходов решения проблемы несоответствия объектной и реляционной модели данных является использование подхода ручного кодирования уровня сохраняемости. В данном случае разработчик должен вручную реализовать методы загрузки/выгрузки данных использую запросы SQL, а также низкоуровневые интерфейсы API для доступа к базе данных (например, ADO. NET для Microsoft. NET или JDBC для Java). Преимущества данного подхода включают возможность достижения высокой производительности (по сравнению с ORM системами) за счет реализации оптимального кода удовлетворяющего требованиям разрабатываемого приложения, а также полный контроль над процессом загрузки/выгрузки данных и возможность использования расширенных возможностей конкретной СУБД. Основным недостатком данного подхода является большое количество рутинной и подверженной ошибкам ручной работы, как на этапе разработки, так и во время сопровождения информационной системы.

Другой подход заключается в использовании систем объектно-реляционного отображения (ORM систем) которые автоматически осуществляют преобразование объектного представления данных в реляционное и наоборот, используя формальное описание отображений [5]. Такое отображение может быть автоматически создано как из существующей базы данных, так и на основе объектной модели данных. Кроме того, отображение может быть описано вручную, использую язык XML подобный язык для описания конфигурация отображения. Популярные ORM решения включают Hibernate для Java [5], и NHibernate и Entity Framework [6] для. NET.

Преимущества ORM решений заключается в значительном сокращение ручной работы связанной с написанием программного кода для доступа к базе данных, а также маппинга реляционных объектов и методов в объектный вид. Наряду с этим ORM решения поддерживают элементы динамического конструирования SQL запросов, что позволяет абстрагироваться от поставщика баз данных (т. е. использовать любой, который поддерживается конкретной ORM системой). Основным недостатком данных систем, является накладные расходы связанные с производительностью. Алгоритмы построения динамических SQL запросов, часто не являются оптимальными. Чтобы решить данные проблемы, разработчики зачастую прибегают к частичной обработки данных внутри приложения: выбирают коллекции объектов и в циклах фильтруют или, используя тот же LINQ над обрабатываемым массивом, порождая новые запросы. Количество таких запросов к СУБД при такой обработке может исчисляться тысячами.

Также к недостаткам ORM систем можно отнести и ограничения, накладываемые ими на разработку программного кода, такие как:

- наследование от базового класса, предоставляемого ORM системой;
- реализация интерфейсов предоставляемых ORM системой;
- ограничение на типы коллекций и ассоциаций.

Техника реляционно-объектного преобразования

В этом разделе мы опишем технику представление реляционно-объектного преобразования, которая может быть использована для решения проблемы преобразования данных из реляционного в объектно-ориентированный вид и наоборот.

Основной задачей реляционно-объектного преобразования является достижение максимальной производительности при взаимодействии приложения с сервером баз данных, а также достижения высокого уровня автоматизации, путем генерации программного кода.

Основной принцип реляционно-объектного проектирования заключается в построении информационных систем на основе модели, описанной и спроектированной на сервере баз данных. Была предложена концепция основанная на построении высокопроизводительной структуры базы данных которая отображает разработанную модель информационной системы. Предоставляя разработчику полный контроль и управление базой данных (построение индексов, написание нетривиальных хранимых процедур, использование представлений и т. д.), мы решаем один из важнейших вопросов связанным с производительностью. Наряду с этим, использование высокоуровневого декларативного специализированного языка SQL, относящегося к четвертому поколению языков программирования (в отличии от C# и Java, относящихся к третьему поколению императивных языков

программирования), позволит оптимизировать процесс обработки данных, и производительность системы в целом, по сравнению с обработкой больших массивов данных внутри приложения.

В процессе проектирования и построения структуры базы данных можно автоматизировать процесс создания так называемых, CRUD хранимых процедур. Структура построения CRUD хранимых процедур имеет строгую типизацию, и отлично укладывается в процесс автоматизации, без риска потери производительности.

Для установления соответствия между реляционной и объектной моделями данных будет использована методология "трех проекций" [7]. Эта методология описывает правила преобразования реляционных объектов в объектную форму и наоборот. Четкое определение реляционных объектов и их свойств позволяет представить их в объектно-ориентированном виде. Данный процесс автоматизируется с помощью использования текстовых шаблонов времени выполнения, таким образом, что процесс генерации программного кода происходит автоматически, на основе спроектированной структуры базы данных.

Таким образом процесс генерации кода уровня сохраняемости приложения полностью автоматизирован. На выходе, разработчик получает набор методов и свойств для взаимодействия клиентского приложения с сервером баз данных. Реализация данной методики положена в основу системы кодогенерации C-Gen [8].

Сравнительная характеристика ORM и C-Gen

В данном разделе на примере модели хранения пользовательских сообщений мы разберем две реализации, первую с использование Entity Framework 5 Code First [9, 10] и вторую с использование системы кодогенерации C-Gen.

Модель данных. В качестве примера будет использована упрощенная модель хранения пользовательских почтовых сообщений (рис. 1). Структура организации и хранения данных следующая:

- Каждый пользователь может иметь несколько папок с письмами;
- В каждой папке может храниться несколько цепочке писем, при этом одна цепочка может храниться в нескольких папках одновременно;
- Каждая цепочка может состоять из нескольких писем;
- Каждая цепочка может быть доступной любому пользователю, в то время как письма – нет.

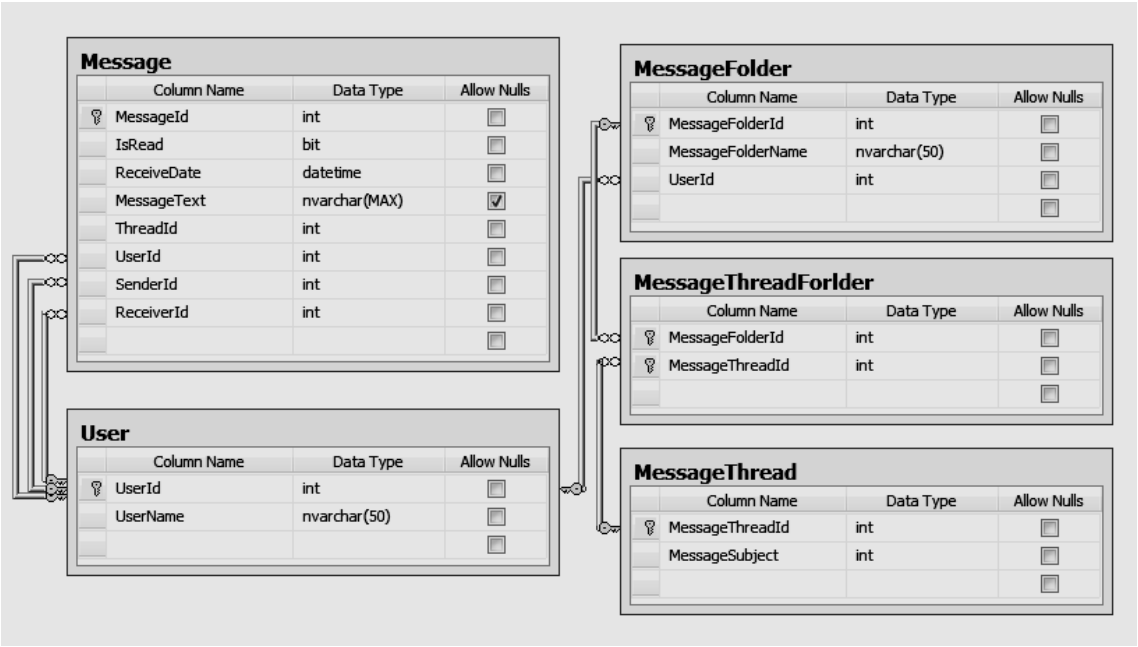

Рис. 1. Диаграмма базы данных по хранению пользовательских почтовых сообщений

Entity Framework

Создание модели данных. В случае уже имеющейся базы данных Entity Framework может автоматически создать модель данных, состоящую из классов и свойств, соответствующих объектам базы данных (таким, как таблицы и столбцы) (рис. 2). Информация о структуре базы, модель данных и маппинг их друг на друга содержится в XML в файле .edmx.

Вне зависимости от наличия базы вы можете вручную написать код классов и свойств, соответствующих сущностям в базе и использовать этот код с Entity Framework без использования файла .edmx. Маппинг между объектами базы данных и моделью данных приложения осуществляется на основе соглашений с помощью специального API. Если базы ещё нет, Entity Framework может создать, удалить или пересоздать её в случае изменений в модели.

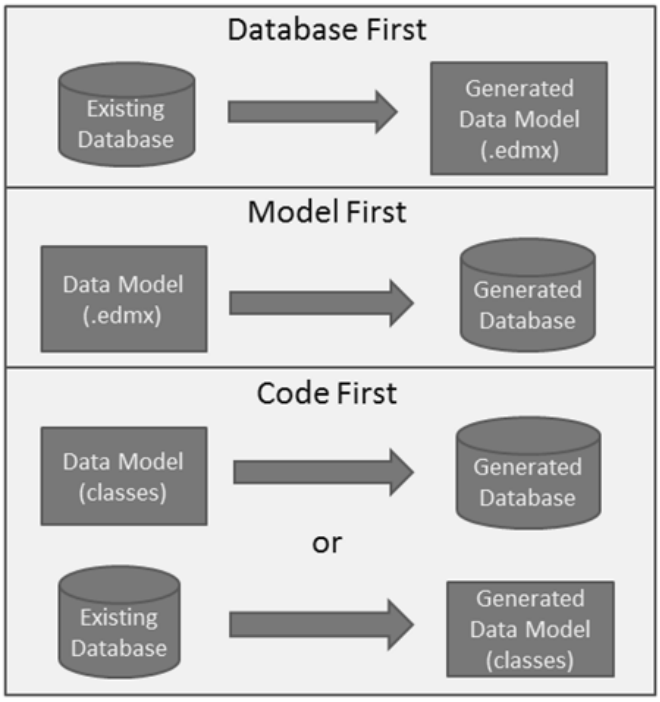

Рис. 2. Подходы к работе с данными в Entity Framework

API доступа к данным, разработанное для Code First, основано на классе DbContext [11]. API может быть использован также и в процессе разработки с подходами Database First и Model First.

Выборка данных. Основной операцией в любой информационной систем является операция выборки данных. В Entity Framework все загруженные объекты попадают во внутренний кэш контекста, затем могут быть запрошены из него напрямую (через метод *Find*).

В приведенном примере (рис. 3) метод *AsNoTracking* отключает кэширование в Entity Framework за получаемых объектов; метод *Include* – задает связанные объекты, включаемые в результаты запроса, через генерацию SQL конструкций в виде *INNER JOIN*; все остальное – обычный LINQ запрос.

```
context.Messages.AsNoTracking().OrderBy(f => f.MessageId).Take(count).Include(f => f.OwnerId.Include(f => f.SenderId).Includ
e(f => f.ReceiverId).ToList();
```

Рис. 3. Запрос на чтения данных в Entity Framework

В Entity Framework любой запрос кроме метода *Find* обращается к базе данных. Метод *Find* выполняет запрос к базе данных лишь в том случае, если объект с необходимыми ключевыми полями не был найден в локальном кэше контекста. Таким образом, из двух вызовов *context.Message.Find(1)* в одном контексте, лишь первый вызов выполнит запрос к БД, тогда как второй уже получит данные из кэша. То же самое справедливо и для загрузки связанных сущностей. Например, в приведенном ниже коде (рис. 4) владелец сообщения будет получен из кэша.

```
context.User.Load();
var m = context.Messages.First();
Console.WriteLine(m.User.Name);
```

Рис. 4. Получение данных из кэша в Entity Framework

Таким образом, выполнение конструкции *context.Message.Include(f=>f.Owner).First()* для того, чтобы предотвратить ленивую загрузку, лишь снизит производительность. Таким образом, использование метода Include неизбежно обрекает разработчиков постоянно следить за тем, какой SQL генерирует Entity Framework, так как иначе это может серьезно повлиять на производительность приложения.

Добавление данных. Добавление данных реализовано довольно просто. Любой объект, добавленный через метод Add, просто добавляется во внутренний кэш контекста со статусом *Added*. При последующем вызове метода *SubmitChanges* проходит проверка данных на соответствие, и для всех подобных сущностей генерируются SQL запрос *INSERT* которые осуществляет добавление объектов в базу данных.Важно, что все связан-

ные свойства навигации, которые не были явно добавлены в контекст, так же принимаются как *Added* и добавляются в базу данных, что может послужить источником трудно отлавливаемых ошибок.

Модификация и удаление данных. При модификации и удаление данных, одной из главных проблем в Entity Framework является отсутствие поддержки Bulk-операций [12]. То есть, то что в SQL можно легко выразить через один запрос, Entity Framework вам понадобиться сделать n запросов, где n – количество записей которые должны быть изменены. Предположим, что в нашем примере необходимо пометить все сообщения в цепочке как прочитанные. В Entity Framework данный запрос будет выглядеть следующим образом (рис. 5):

```
var messages = context.Message.Where(f=>f.MessageThread.Id == threadId);
foreach (var m in messages)
    m.IsRead = true;
context.SaveChanges();
```

Рис. 5. Обновление данных в Entity Framework

В результате выполнения мы получим:

- будет выполнена загрузка всех модифицируемых записей в контекст, а затем генерация для каждой из них отдельного;
- большое количество сущностей в памяти, повлечет за собой накладные расходы по освобождению этой памяти, а большое количество запросов создаст дополнительную нагрузку на сервер БД;
- кроме того, существуют некоторые проблемы с проверкой данных на соответствие (валидацией), которые будут описаны далее.

Операция удаления данных аналогична по своей структуре обновлению и имеет те же проблемы с производительностью.

Коллекции как свойства навигации. Рассмотрим следующий пример: Предположим, что у нас есть некий список папок сообщений. Возле имени каждой папки необходимо отображать количество цепочек находящихся в данной папке. Для реализации этой функциональности зачастую используется следующий запрос (рис. 6):

```
var count = folder.Threads.Count();
```

Рис. 6. Обход коллекции в Entity Framework

Особенность Entity Framework (а также и LinqToSql) заключается в том, что, при вызову метода *Count*(), или *Any*(), либо выполнения запроса на коллекции используемой в качестве навигационного свойства, происходит к полной загрузке коллекции в память.

Атрибут Required. В Entity Framework атрибут *Required* указывает на то, что поле в базе данных не может быть равно *NULL*. В нашем примере в объекте *Message* такие свойства как *Owner*, *Sender*, *Receiver* и *Thread* помечены как *Required*. Помимо того, что это указывает Entity Framework на то, что поля должны быть помечены как *NOT NULL*, включает на них внутреннюю проверку Entity Framework. Рассмотрим следующий пример, который устанавливает статус сообщения как прочтенное (рис. 7).

```
var message = context.Messages.Find(1);
message.IsRead = true;
context.SaveChanges();
```

Рис. 7. Особенности работы с контекстом Entity Framework.

Выполнение данного запроса при включенной валидации на только что созданном контексте приведет к появлению исключения. *Required* атрибут генерирует ошибку валидации, ведь все помеченные им поля были равны *NULL*, а выполнить ленивую загрузку их он не умеет.

Система кодогенерации C-Gen

Создание модели данных. Система кодогенерации C-Gen в качестве модели принимает готовую базу данных, на основе которой в программном коде генерируются набор соответствующих объектов и методов, которые отображают маппинг структуры базы на объектную модель. Наряду с маппингом реляционных объектов базы данных на объектную модель, в самой базе данных генерируются CRUD хранимые процедуры, через которые осуществляется манипулирование данными (рис 8).

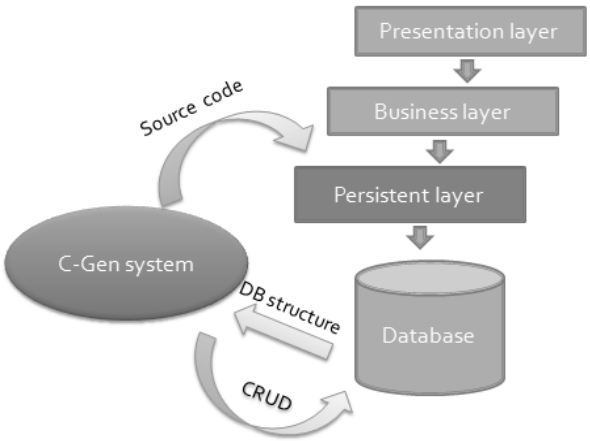

Рис. 8. Схема взаимодействия компонентов информационной системы с системой кодогенерации C-Gen

Выборка данных. Чтобы осуществить выборку данных, с использованием API системой кодогенерации C-Gen, доступно два варианта:

- воспользоваться одним из методов сгенерированной системой автоматически;
- если функциональности стандартных запросов не достаточно, реализовать SQL запрос в виде хранимой процедуры, и осуществить перегенерацию API. В результате из кода будет доступен метод который осуществит выборку данных и вернет объект.

Для реализации вышеуказанной выборки в C-Gen, необходимо реализовать хранимую процедуру, выполняющую выборку данных (рис. 9).

```
 8
 9  create PROCEDURE [dbo].[Message_GetMessageList]
10      @Count int
11  AS
12      select top(@Count)
13          [m1].[MessageId],
14          [m1].[IsRead],
15          [m1].[ReceiveDate],
16          [m1].[MessageText],
17          [u1].[UserId], -- Owner Id
18          [u1].[UserName], -- Owner Name
19          [u2].[UserId], -- Sender Id
20          [u2].[UserName], -- Sender Name
21          [u3].[UserId], -- Receiver Id
22          [u3].[UserName] -- Receiver Name
23      from [dbo].[Message] [m1]
24      inner join [dbo].[User] as [u1] on [m1].[UserId] = [u1].[UserId]
25      inner join [dbo].[User] as [u2] on [m1].[SenderId] = [u2].[UserId]
26      inner join [dbo].[User] as [u3] on [m1].[ReceiverId] = [u3].[UserId]
27      order by [m1].[MessageId] asc
```

Рис. 9. Листинг хранимой процедуры GetMessageList

Сгенерированный API предлагает на выбор два метода, для получения результатов выборки:

- первый метод в результате выборки вернет объект типа DataReader, из которого в дальнейшем можно вывести типизированный объект (подразумевается написание класса обертки). Данный метод подходит, если результирующий объект будет использован всего один раз и в процессе выборки, необходимо реализовать дополнительную логику по агрегации данных. Недостатком же данного подхода является то, что разработчик фактически работает динамическим объектом, и в результате изменения структуры возвращаемых данных на стороне сервера баз данных, может привести к ошибке времени выполнения приложения, если пользователь не внесет изменения в клиентском приложении (исключительная ситуация может возникнуть в случае переименования или удаления одного из полей входящих в результат выборки).
- второй метод основан на алгоритма **"рекурсивного выведения типа"**. В программном коде будет сгенерирован строго-типизированный объект (в виде класса), отображающий результат выборки. Использовать подобный класс удобно, т. к. не нужно производить дополнительных операций по выведению из DataReader класса соответствующего результату выборки. Однако если в процессе разработки, возникнет необходимость в написании другого метода (SQL запроса), результирующим набором которого должен будет стать объект с теми же полями и свойствами, система создаст для него новый класс, вместо того чтобы использовать уже существующий.

Для решения проблемы работы в API со строго-типизированными объектами, существует механизм выведения типа объекта через представление. Данный подход подразумевает написание представления, которое будет является отображением результата выборки (рис. 10). Таким образом будет сгенерирован метод, результатом которого является класс построенный на основе полей представления.

```
 6  create View [dbo].[vMessageList]
 7  AS
 8      select
 9          [m1].[MessageId],
10          [m1].[IsRead],
11          [m1].[ReceiveDate],
12          [m1].[MessageText],
13          [u1].[UserId], -- Owner Id
14          [u1].[UserName], -- Owner Name
15          [u2].[UserId] as [SenderId], -- Sender Id
16          [u2].[UserName] as [SenderName], -- Sender Name
17          [u3].[UserId] as [ReceiverId], -- Receiver Id
18          [u3].[UserName] as [ReceiverName] -- Receiver Name
19      from [dbo].[Message] [m1]
20      inner join [dbo].[User] as [u1] on [m1].[UserId] = [u1].[UserId]
21      inner join [dbo].[User] as [u2] on [m1].[SenderId] = [u2].[UserId]
22      inner join [dbo].[User] as [u3] on [m1].[ReceiverId] = [u3].[UserId]
23  GO
```

Рис. 10 Листинг представления vMessageList

Данный подход, предполагает агрегацию данных в представлении, а затем выборку их по средствам хранимой процедур (рис. 11), однако позволяет использовать один и тот же класс в качестве результата выборки для нескольких хранимых процедур.

```
 8
 9  create PROCEDURE [dbo].[vMessageList_GetMessages]
10      @Count int
11  AS
12      select top(@Count)
13          [m].[MessageId],
14          [m].[IsRead],
15          [m].[ReceiveDate],
16          [m].[MessageText],
17          [m].[UserId],
18          [m].[UserName],
19          [m].[SenderId],
20          [m].[SenderName],
21          [m].[ReceiverId],
22          [m].[ReceiverName]
23      from [dbo].[vMessageList] [m]
24      order by [m].[MessageId] asc
25
```

Рис. 11. Листинг запроса GetMessage

В результате, мы получаем метод, который возвращает объект, отображающий результат выборки SQL запроса. Важно понимать, что при взаимодействии со сгенерированным API, разработчик реализовывает только SQL на стороне сервера баз данных, все остальные классы и методы доступа, генерируются автоматически. Таким образом можно достичь высокой производительности за счет использования оптимальных запросов со стороны сервера баз данных, а также высокий уровень автоматизации, за счет генерации API доступа и CRUD автоматически.

Добавление данных. Для того чтобы добавить данные в сгенерированном API необходимо создать объект того типа, данные которого необходимо добавить, а затем выполнить метод *Save*(). Важно отметить, что при необходимости выполнить несколько методов в разрезе одной транзакции, такая возможность существует. Для этого необходимо создать экземпляр класс *TransactionManager,* а в каждый метод, находящийся в транзакции необходимо передать экземпляр созданного класса (рис. 12).

Таким образом в результате возникновения исключительной ситуации, вызов метода *Rollback*() не внесет изменений в пределах цепочки вызовов методов, помеченных транзакцией.

Модификация и удаление данных. В отличии от Entity Framework, API сгенерированной системой кодогенерации C-Gen поддерживает Bulk-операции, а это значит, что на примере с перебором коллекции и установки в ней писем как прочтенных выполнится всего два запроса к базе данных: первый вернет список всех сообщений. Далее необходимо пройти по списку у становить свойство *IsRead = true*, а затем выполнить метод Save() в который передать коллекцию сообщений для сохранения. Тоже справедливо и при удаления данных.

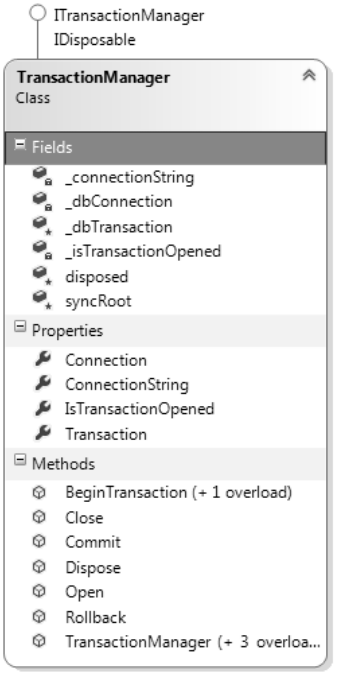

Рис. 12. Диаграмма класс *TransactionManager*

Выводы

В данной статье была рассмотрена проблема преобразования данных из объектно-ориентированного вида в реляционный и наоборот. Один из наиболее распространенных подходов реализации такого взаимодействия: использование ORM систем. Были проанализированы преимущества и недостатки использования ORM систем, на основе чего были сформированы требования, которым должна удовлетворять система эффективного преобразования данных из объектного представления в реляционное и наоборот.

Исходя из этого, предложена методика реляционно-объектного представления, которая базируется на принципах построения информационных систем на основе модели в базе данных. Данная методика реляционно-объектного отображения решает две основные проблемы, характерные для систем подобного рода:

- **производительность.** Оптимальные запросы, написанные SQL разработчиком, оказываются на много более эффективными чем генерируемый ORM SQL код;
- **автоматизация.** Алгоритм генерации программного кода, основанный на методике «трех проекций» позволяет установить соответствие между объектным и реляционным представлением данных, а также предоставить разработчику API для доступа к данным.

Данная методика, положена в основу системы кодогенерации C-Gen. Сравнительная характеристика, решения проблемы объектно-реляционного преобразования на практическом примере показала, эффективность работы системы кодогенерации C-Gen по сравнению с Entity Framework. Основным недостатком использования Entity Framework , по сравнению с системой кодогенерации C-Gen является неоптимальный SQL код генерируемый системой. Разработчик должен постоянно следить за тем, какой SQL код был сгенерирован системой. Таким образом, без базовых знаний основ реляционной алгебры, невозможно построить систему, оптимальную по производительности (хотя предлагаемый подход подталкивает разработчика абстрагироваться от написания SQL, доверив это ORM системе). При осуществлении операций добавления коллекций данных, ORM не поддерживает Bulk операции, поэтому на каждом витке итерации необходимо выполнять запрос к серверу баз данных.

1. *Russell C.* Bridging the Object-Relational Divide. Queue Vol. 6, N 3, May 2008, P. 18–28.
2. *Создание* кода и текстовые шаблоны T4 http://msdn.microsoft.com/ru-ru/library/bb126445.aspx.
3. *Booch G., Maksimchuk R.A., Engle M.W.* Object-Oriented Analysis and Design with Applications. Addison-Wesley, 2007/
4. *Russell C.* Bridging the Object-Relational Divide. Queue. – Vol. 6, N 3. – May 2008, P. 18–28.
5. *Bauer C., King G.* Java Persistence with Hibernate. Manning, New York, 2007.
6. *O'Neil E.J.* 2008. Object/relational mapping 2008: hibernate and the entity data model (edm). In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08). – 2008. – P. 1351–135.
7. *Лихацкий И.А.* Об одной методике формирования объектного представления реляционных данных // Проблеми програмування. – 2013. – № 3. – С. 79–85.
8. *Лихацкий И.А.* Средства кодогенерации для взаимодействия с базой данных через объекты // Проблеми програмування. – 2012. – № 2–3. – С. 384–385.
9. *Учебный* курс. Создание модели данных Entity Framework для приложения ASP.NET MVC http://habrahabr.ru/company/microsoft/blog/133316/
10. *Еще* один взгляд на Entity Framework: производительность и подводные камни http://habrahabr.ru/post/164483/
11. *DbContext* Class http://msdn.microsoft.com/en-us/library/system.data.entity.dbcontext%28v=vs.113%29.aspx
12. *BULK* INSERT (Transact-SQL) http://msdn.microsoft.com/en-us/library/ms188365%28SQL.100%29.aspx