Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Применение генеративного программирования для создания объектной базы

Дипломная работа студента 544 группы

Никитина Павла Антоновича

Научный руководитель	.....	канд. техн. наук В.С. Гуров
	/подпись/	
Рецензент	.....	ст. преп В.С. Полозов
	/подпись/	
“Допустить к защите”	.....	д.ф.-м.н., проф. А.Н. Терехов
Заведующий кафедрой	/подпись/	

Санкт-Петербург

2010

St. Petersburg State University

Faculty of Mathematics and Mechanics

Chair of Software Engineering

Object database implementation with generative programming

Graduate paper of student of group 544

Pavel Nikitin

Academic advisor	..... /signature/	candidate of technical sciences Vadim Gurov
Reviewer	..... /signature/	Senior lecturer Victor Polozov
"Admitted to defense" Chair leader	..... /signature/	Professor Andrey Terekhov

St. Petersburg

2010

Содержание

1 Введение

При разработке программных систем возникает необходимость долговременного хранения данных. Наиболее распространённым на данный момент способом для этого являются реляционные базы данных, основанные на реляционной парадигме, базирующейся на хорошо зарекомендовавших себя математических (алгебраических) принципах. Соответствующие системы позволяют хранить информацию в таблицах и манипулировать с ней при помощи специального языка внутри базы, либо с внешними вызовами. С другой стороны, современная практика программной инженерии базируется на объектно-ориентированном подходе, в котором объекты обладают как данными, так и поведением. Отличие между этими моделями получило название paradigm mismatch (несоответствие парадигм) или impendance mismatch (что можно вольно перевести как несоответствие выразительности). Действительно, в объектном подходе отношения выражаются через обход объектов по ссылкам, в то время, как в реляционном подходе — через объединение таблиц. Безусловно, при достаточном понимании обеих моделей, между ними можно провести много параллелей и аккуратно реализовать интеграцию, но проблема несоответствия ведёт к усложнению разработки и поддержки таких программ.

В противовес смешению многих моделей существует концепция, предполагающая, что языки общего назначения должны обладать расширенными средствами манипуляции с данными, опираясь на теорию и устоявшуюся практику работы с операциями, образующими систему запросов над коллекциями на основе замыканий [10], пришедшую из функциональных языков. Существующие коллекции в языке Java (версия 6) не реализуют подобный механизм (более того даже наличие функциональных типов в следующей версии платформы (версия 7) стоит под вопросом [6]). В этом смысле Java является весьма консервативным языком, что приводит ко всевозможным попыткам реализации внутренних DSL (domain-specific languages, предметно-ориентированные языки). Однако внутренние DSL весьма близки к языкам программирования, на основе которых построены (являясь выразительной обёрткой тех или иных их конструк-

ций), поэтому на них нельзя выразить то, для чего нет средств в исходном языке. Внешние же DSL не решают проблему символьной интеграции. Кроме того, работа с DSL без средств поддержки в среде разработки (IDE) предполагает от пользователя как минимум, тщательного освоения нового синтаксиса. Под поддержкой в IDE подразумевается возможность проведения статического анализа кода, проверки используемых выражений на соответствие его синтаксису и системе типов, анализ потока данных, рефакторинг [9] и т. п. Если же DSL превносит свою парадигму — то и понимание взаимодействия парадигм. Для некоторых из существующих решений существует поддержка в IDE, но она не является универсальной и предоставляется поставщиком IDE, не связаным напрямую с поставщиком технологии, что замедляет внедрение и исправление ошибок. Внедрение интерпретируемых внутренних DSL без учёта системы типов (формулирование запросов внутри строковых литералов, например) приводит к невозможности выявления синтаксических ошибок кодирования на стадии компиляции.

Обобщим возможные характеристики для классификации технологий хранения данных в объектных языках:

- совместимость с объектной парадигмой
- поддержка со стороны IDE
- поддержка со стороны системы типов

Различные системы можно характеризовать степенью соответствия этим характеристикам модели собственно хранения объектов и системы запросов над ними. Все эти свойства относятся не к собственно функциональности, а к производительности труда, что является приоритетной задачей в данной работе, поэтому решение предполагает соблюдение всех вышеприведённых условий без существенной потери производительности программы. Оказалось, что такие решения, не вовлекающие инструментирование байт-кода (анализ промежуточного исполнимого представления кода непосредственно во время работы программы), не получили распространения. Использование

инструментирования, в свою очередь, представляется серьёзным усложнением, которое предполагается обойти средствами генеративного программирования. Также будет приятно во внимание то, что с отказом от реляционной парадигмы теряются многие средства поддержания целостности данных, которые отсутствуют в объектной модели. Разработка решения велась с ориентацией на конкретный проект (многопользовательское веб-приложение со сложной схемой данных), что обеспечило достаточную полноту его функциональности без ущерба универсальности. На данный момент, одним из немногих средств, позволяющим создавать совместимые между собой языковые расширения вместе с необходимой инфраструктурой для них, является JetBrains MPS, которая была выбрана для реализации расширения языка Java для работы с объектной моделью хранения данных.

Выбранная тема работы актуальна как с точки зрения работы с хранимыми данными, так и с точки зрения актуальности проблемы производительности труда при работе с предметно-ориентированными языками, что будет показано далее.

2 Обзор

Ниже приведен обзор основных подходов к работе с хранимыми данными на языке Java. Сначала будут рассмотрены не совместимые с объектной парадигмой решения, затем совместимые частично и, наконец, совместимые полностью. Каждый из подходов будет оценен с точки зрения выразительности и степени интеграции выразительных средств, степени предотвращения ошибок кодирования на ранних стадиях разработки приложения. Дополнительно, будет рассмотрена эффективность использования того или иного подхода в рамках трёхуровневой модели, предполагающей деление кода приложения на уровень представления, уровень бизнес-логики и уровень хранения [17]. Рассмотрим данную модель абстрактно — с точки зрения инкапсуляции и разделения функциональности:

1. уровень представления — описывает и реализует сценарии отображения данных и их интерактивного изменения пользователями

2. уровень хранения — реализует преобразование хранимых данных в объекты, поиск и навигацию по ним

3. уровень бизнес-логики — средний, связывает различные компоненты приложения, реализует специфичную для приложения валидацию данных, реализует взаимодействие между соседними уровнями

2.1 Простейший подход

JDBC [20] — платформенно-независимый стандарт взаимодействия Java-приложений с различными СУБД, реализация которого является частью платформы (пакет java.sql) практически с самого начала её существования (версия 1.1). Программный интерфейс JDBC инкапсулирует сетевой протокол взаимодействия с внешней базой данных в плане отсылки SQL-запросов, итерирования по полученным данным с помощью курсоров и работы с транзакциями. Библиотека предоставляет примитивы для подготовки, использования и переиспользования запросов на основе данных,

доступных во время исполнения программы, с помощью конкатенации строк, что позволяет динамически формировать эффективные запросы. Для результатов запросов доступны их метаданные (количество колонок, и т. п.) и прямая адресация по их табличному представлению. Средства отображения объектов на таблицы отсутуствуют.

Как видно, JDBC является простейшим адаптером между объектным языком и базой данных, не предоставляя никаких средств для ухода от реляционной парадигмы. Сколько-нибудь существенная поддержка со стороны распространённых IDE отсутуствует, и её реализация в общем виде представляется проблематичной, как минимум из-за того, что задача проверки корректности программы, работающей со внешними данными является в общем виде неразрешимой, а JDBC позволяет строить запросы на основе любых данных, имеющихся в распоряжении програмы. Кроме того, система типов, описывающая интеграцию между Java и SQL, как таковая отсутствует и описана в документации к стандарту как внешнее знание, т. е. программный интерфейс никак не ограничивает множество операций над данными в зависимости от их типа.

При использовании JDBC реализация уровня хранения вынуждена опираться на конкретную схему данных, тем самым исключая переиспользование существующего кода в рамках цикла разработки (что увеличивает плотность ошибок, количество переделок и уменьшает продуктивность, как показано в [4]).

2.2 Объектно-реляционное проецирование

Наиболее распространённым подходом является объектно-реляционное проецирование (object-relational mapping, ORM [12]), осуществляющее адаптацию между объектной и реляционной парадигмой. При этом выполняется автоматизированное и прозрачное хранение Java-объектов в таблицах реляционной базы данных. Преобразование между представлениями задаётся метаданными. ORM реализует набор API (программных интерфейсов):

- интерфейс для выполнения CRUD-операций (создание, чтение, модифицирование, удаление) над объектами хранимых классов

- интерфейс или язык для построения запросов на основе классов и их свойств
- способ описания метаданных
- способ работы с транзакциями и оптимальным обходом данных в их рамках

ORM позволяет разрешить несоответствие парадигм в смысле хранения объектов и свойств, но не скрывает реляционную сущность ассоциаций. Благодаря тому, что работа с данными (CRUD) осуществляется в памяти без переопределения семантики Java, поддержка со стороны IDE и безопасность системы типов остаётся на уровне самого языка. Реализация хранения метаданных возможна как в форме внешнего DSL (описание метаданных в XML-файлах, например), так и в форме внутреннего DSL (аннотации JPA, стандартизованы в результате деятельности группы JSR 220, с такими релизациями, как TopLink). Недостаток первого подхода следует из недостатка комбинирования основного языка с внешним DSL: отстутсвие символьной интеграции, т. е. обоюдной интеграции элементов AST одного языка в другой для их взаимодействия.

К сожалению, языки запросов в канонических реализациях ORM, таких, как Hibernate [5], остаются реляционными и оформлены как внутренние DSL над строковыми литералами (подобно прямому встраиванию SQL в JDBC). Опытные пользователи таких систем признают, что детальное понимание реляционной парадигмы необходимо для реализации оптимальных запросов: "To use Hibernate effectively, a solid understanding of the relational model and SQL is a prerequisite [...] Hibernate will automate many repetitive coding tasks, but your knowledge of persistence technology must extend beyond Hibernate itself if you want take advantage of the full power of modern SQL data-bases."

С поддержкой внутренних DSL запросов обстоит лучше: благодаря распространности Hibernate и стандартизации JPA ведущие на рынке IDE обладают поддержкой этих средств и проверкой типов, однако полнота и полное соответствие модели может разниться в зависимости от реализации.

Поскольку такие решения, как Hibernate выполнены в форме промежуточного программного обеспечения, реализация уровеня хранения, целиком покрываемая ими, может считаться полностью переиспользуемой, что улучшает качество продуктов, созданных с их помощью [4].

Из-за того, что стандартные, как Java, объектные языки не являются ленивыми [14] в смысле внутреннего состояния объекта класса, производительность преобразования из реляционного представления в объектное может страдать из-за невозможности частичной загрузки данных. Даже Hibernate, изначально не предполагавший инструментирование, пришёл к его необходимости для реализации ленивой загрузки [1], изменяя семантику обращения к полям объекта, причём использование инструментирования накладывает ограничение на использование модификатора **final** в объявлении хранимого класса и его методов.

2.3 Объектные базы

Идея хранить непосредственно те данные, с которыми манипулирует программа, привела к формированию направления объектных баз, которые в простейшем виде существовали ещё до популяризации реляционных, но начинают составлять им конкуренцию и обретать законченность они сравнительно недавно. В качестве примера объектной базы данных рассмотрим db4o [19]. Дальнейшим развитием технологии инструментирования являтся его использование для оптимизации запросов, реализованное в этой системе. По сравнению с слабо типизированным языком запросов на основе реляционной парадигмы сделан шаг вперёд: запросы формулируются на Java, в рамках собственно объектной модели, без отображения в реляционную. При этом авторам db4o пришлось отказаться от реляционной модели и перейти на объектные понятия. Таким образом достигается полная совместимость с объектной парадигмой. Семантика запросов на Java не переопределяется, что минимизирует необходимость изучения новых понятий. Инструментирование позволятет восстанавливать дерево запроса про сгенерированному компилятором байт-коду и осуществлять всевозможные

оптимизации в зависимости от контекста.

2.4 Целостность данных

Недостатком хранения объектной модели из Java является отсутствие распространённых в реляционных базах средств поддержания целостности данных [24]. Ниже будет приведён список таких средств и проведён обзор их применимости в рамках объектной модели.

- первичные ключи
- внешние ключи
- условия на простые типы (ограничение домена возможных значений)
- условия на ссылочную целостность
- триггеры
- хранимые процедуры

2.4.1 Условия на ссылочную целостность

К сожалению, добиться прозрачного переноса объектной модели на хранимые данные невозможно. Семантика объектной модели памяти как для управляемых (C#, Java), так и неуправляемых языковых сред не вполне подходит для них. В неуправляемых средах с явным удалением объекта семантика разыменовывания ссылки на удалённый объект не определена (C++). В управляемых средах операция явного удаления объекта отсутствует, вместо чего присутсвует политика сборки мусора: весь подграф объектов без входящих ссылок (из корневых объектов, к которым можно отнести статику и локальные переменные) считается подлежащим удалению, что не приемлемо для хранимых объектов, так как нет естественного определения корневых объектов (не подлежащих удалению без входящих ссылок), и даже несвязность хранимого графа не является достаточным условием для удаления множества объектов.

В db4o множество возможных действий над объектом дополнено операцией удаления, причём разыменовывание ссылки на удалённый объект даёт null, что, в каком-то смысле, нарушает ссылочную целостность. Также в db4o возможно выполнение простейших компенсирующих операций, описанное в [24] — рекурсивное удаление объектов, которые содержат ссылку на удаляемый. При этом отсутствует возможность задания модификатора RESTRICT, запрещающего удаление объекта, на который ссылаются другие.

2.4.2 Внешние и первичные ключи

В реляционной модели данных ссылки между объектами (отношениями) реализованы следующим образом: задаётся упорядоченная пара множеств атрибутов одного типа в двух отношениях, причём первый атрибут называется внешним ключом, а отношение, его содержащее по смыслу можно считать началом ссылки. Такие ссылки являются достаточно гибкими, но неустойчивы к изменениям схемы. В объектной модели каждый объект *неявно* обладает уникальным идентификатором, соответственно, понятие первичных ключей присутствует (в видоизменённом виде). Отличие заключается в том, что в объектной модели понятие состояния отделено от понятия объекта: разные объекты могут иметь одинаковое состояние. В реляционной же модели сущность полностью определяется своим состоянием.

2.4.3 Триггеры

В объектной модели аналогом триггеров являются слушатели (listeners). Существуют решения (такие, как ODE [13]), расширяющие семантику объектного языка (C++) абстракциями для поддержания целостности данных, в том числе и триггерами, основаными на предикатах. Сама идея использования триггеров не нова [3, 15], примером триггера являются действия, происходящие в реляционной базе при указании CASCADE. В [7] под триггером понимается особый вид хранимой процедуры, вызываемой при модификации определённой таблицы, на объектную модель это можно

естественно перенести как особый метод (см. ниже), вызываемый при модификации хранимого объекта определённого типа, хотя в ODE триггеры привязываются не к типам объектов, а к самим объектам.

2.4.4 Хранимые процедуры

Как таковые, хранимые процедуры не применимы в случае встроенной базы данных без разделения окружений на клиентское и серверное. Однако следует отметить, что в неявной форме их аналог возможен и в объектной модели. Дело в том, что дополнительная роль хранимых процедур в традиционных системах управления базами данных, помимо улучшения производительности за счёт прекомпиляции, например — это инкапсуляция. Инкапсуляция, очевидно, имеет прямое отношение к целостности данных, которые требуют соблюдения нетривиального протокола при работе с ними. Соответственно, даже в обычном ООП существует аналог хранимой процедуры в смысле инкапсуляции — метод, работающий с полями (данными) с ограничивающими прямой доступ модификаторами.

В [18] реализована система запросов (LINQ, language integrated query) над коллекциями на основе замыканий и функциональных типов, причём предоставлена возможность заменять реализацию деревьев запросов с целью их оптимизации под различные источники данных. Тем самым решается проблема стандартизации работы с запросами: с одной стороны, они включены в стандарт языка [8], с другой — предоставляют возможность переопределения, поэтому пригодны для реализации объектно-реляционного проецирования (Linq to SQL [18]) или работы с хранимыми объектами (ADO.NET entity framework [2]).

3 Постановка задачи

Резюмируя предыдущую часть, можно заметить, что с точки зрения эффективности программирования среди получивших распространение решений для Java выигрывают те, в которых сделана ставка на уход от реляционной модели, что, в основном, обусловлено отказом от явной языковой интеграции в пользу оптимальных запросов при совмещении реляционной модели с объектной. В решениях без использования дополнительных языков недостаточно проработанной оказывается теоретико-множественная модель хранения объектов, которая, очевидно, не может один в один быть позаимствована из самого языка Java. Существуют решения, в которых эта модель удачно расширена, но выражается в терминах нового языка [13], что влечёт отсутствие IDE-поддержки. Целью данной работы является объединение лучших сторон рассмотренных решений:

1. совместимая с объектной парадигмой расширенная модель хранения данных, содержащая средства поддержания целостности данных
2. декларативное описание метаданных, необходимых для этой модели
3. символьная интеграция между описанием метаданных и основным языком
4. полноценная языковая инфраструктура (IDE)
5. интеграция с системой типов

4 Теоретическая модель объектной базы

В этой части вводятся понятия, используемые для формализации реализованной объектной модели (концептуальный уровень в терминах Дейта).

4.1 Абстрактный уровень

В отличие от реляционных баз, основным понятием в объектной модели являются объекты, а не отношения. Формально, схема базы данных определяется как набор:

1. множество строк (допустимых имён) S, за S^* обозначим множество всех наборов с элементами из S; за S^+ обозначим множество всех наборов с элементами из S, кроме пустого.

2. множество типов объектов (Entity) TY с уникальными именами (биекция $TY \leftrightarrow S$)

3. множество примитивных типов P с областью значений для каждого, в каждой из которых выделено нулевое значение

4. множество примитивных полей для каждого типа $PR = TY \Rightarrow P \times S^*$, проекция которого ($TY \Rightarrow S^*$) обладает свойством функциональности; $\Rightarrow$ используется для обозначения функции с несколькими значениями

5. множество F направленных ссылок между объектами (отображение множества $TY \times TY \Rightarrow S^*$)

Экземпляром базы данных со схемой (S, TY, P, PR, F) называется M — типизированное (с отображением $T = M \Rightarrow TY$) множество объектов с отображением $D = M \times M \Rightarrow S^*$, таким что $\forall s \in S : s \in D(m1, m2)$ тогда $s \in F(T(m1), T(m2))$. $L(m1, m2, s)$ назовём экземпляром ссылки $l(T(m1), T(m2), s)$. Здесь $T(m2)$ — тип ссылки, s — имя ссылки. Также для каждого объекта задано множество значений примитивного типа (аналогично, с привязкой к PR). Формализация примитивных типов в рамках теории множеств опущена строится подобным образом.

Здесь множество строк используется для уникальной идентификации типов и именованных ссылок. С точки зрения концептуального уровня их строковая природа не важна, но она соответствует способу формирования идентификаторов, таких как имена классов и переменных, в текстовых языках программирования. Типы объектов, как видно, не образуют иерархию — они соответствуют конкретным возможным типам объектов, характеризующимся множеством возможных ссылок и примитивных полей (сигнатура типа).

Действительную мощность (*cardinality*) направленной сслыки определяется через набор мощностей множеств экземпляров этой ссылки по всем объектам. Возможные варианты (нижней и верхней границы чисел в этом наборе), все упомянутые множества всегда являются конечными:

- 0..1 (нижняя граница равна 0, верхняя равна 1) — одиночная ссылка
- 1 (верхняя граница совпадает с нижней и равна 1) — обязательная одиночная ссылка
- $0..n$ (нижняя граница равна 0, верхняя больше 1) — множественная ссылка
- $1..n$ (все остальные случаи) — обязательная множественная ссылка

Основной операцией над моделью являются добавление и удаление пар в отображении F. Данная базовая модель (наиболее абстрактный уровень) описывает модель памяти, обобщённо соответствующую модели памяти для объектов в объектно-ориентированных ЯП. Отличие состоит в том, что одна ссылка может иметь несколько экземпляров, отличающихся только концом, то есть обладать действительной мощностью $0..n$ или $1..n$. В обычных же языках без расширений, чаще всего, возможна действительная мощность 0..1 или 1 (но невозможно задать инвариант, гарантирующий 1 вместо 0..1).

4.2 Концептуальный уровень

На концептуальном уровне схема расширяется множеством предикатов, выполнение которых для каждого экземпляра базы с этой схемой должно гарантироваться реализацией СУОБД. Каждый предикат задаётся декларативно, т. е. действует на основе заданных дополнительных множеств/отображений.

1. Дополним схему базы понятием мощности (*cardinality*) ссылки, на абстрактном уровне мощность ссылки является количественным параметром, зависящим от природы конкретного экземпляра. Вспомогательная конструкция: множество возможных мощностей $PWR = \{0..1, 1, 0..n, 1..n\}$, на нём задан частичный рефлексивный порядок: $1 \preceq 0..n; 0..1 \preceq 0..n; 1..n \preceq 0..n, 1 \preceq 1..n; 1 \preceq 0..1$ (рефлексивные пары опущены). Назовём мощности $0..1$ и 1 одиночными, а $0..n$ и $1..n$ — множественными.

   Декларативное описание: отображение $F \Rightarrow PWR$. Таким образом, задание мощности при описании ссылки является обязательным (хотя можно бы было принять $0..n$ за значение по умолчанию, так как $\forall p \in PWR : p \preceq 0..n$).

   Предикат: для ссылки $L = (t1, t2, s)$ с мощностью p и действительной мощностью $p1$ на данном экземпляре базы частичный порядок между p и $p1$ должен быть определён и $p1 \preceq p$. Таким образом, в обычных языках мощность всех ссылок равна $0..1$.

   Примечание: существуют расширения языков, например аннотация @NotNull для Java, добавляющая возможность выполнять статическую проверку на строгое равенство мощности единице.

2. Дополним схему понятием двунаправленной ссылки. Декларативное описание: множество пар $((t1, t2, s1), (t2, t1, s2))$, оба элемента которых являются ссылками в рамках схемы, причём максимум одна из этих ссылок является множественной (т. е. двунаправленная ссылка не является отношением). Неформально, если

экземпляр базы — это ориентированный граф, то экземпляр двунаправленной ссылки обязан быть двунаправленным ребром.

Предикат: для каждого экземпляра ссылки, являющейся частью двунаправленной (декларация содержит соответствующую пару) $l1 = (m1, m2, s1)$ существует противоположный экземпляр $l2 = (m2, m1, s2)$.

3. Дополним схему понятием агрегации. Декларативное описание: множество пар $((t1, t2, s1), (t2, t1, s2))$, оба элемента которых являются ссылками в рамках схемы, причём вторая всегда имеет мощность 0..1. Первая часть ссылки называется ссылкой родитель-ребёнок, а вторая, соответственно, ребёнок-родитель. Кроме того, неформально, каждый тип, у которого могут присутствовать ссылки на родителя (потенциальный ребёнок) должен иметь ровно одного родителя.

   Предикат: ...

4. Дополним схему понятием обязательного примитивного поля. Декларативное описание: подмножество множества примитивных полей, для которых запрещено нулевое значение, например, ограничение на ненулевую длину экземпляра строкового типа.

4.3 Транзакции

Неформально, транзакция — результат последовательного выполнения операций. Операция — преобразование экземпляра базы (M, T, D) в экземпляр (M', T, N') с той же схемой и допустимой релаксацией предикатов концептуального уровня (и любых других, кроме определения экземпляра). Также, операция должна соблюдать естественное условие на сохранение типов: $e \in M' \& e \in M \Rightarrow T(e) = T'(e)$. Рассмотрим множество операций, доступных для применения к (M, D).

1. Создание объекта e с типом t.

- $M' = M \cup \{e\}$

- $T' = T \cup \{e, t\}$

Здесь добавляется объект с указанием его типа.

2. Удаление объекта e с типом t.

- $M' = M/\{e\}$
- $T' = T/\{e, t\}$
- D' — проекция D на $M' \times M'$

При удалении объекта отображение D урезается до его проекции на меньшее множество M' (т. е. удаляются все входящие и исходящие ссылки).

3. Добавление экземпляра ссылки с именем s между $e1$ и $e2$.

- $M' = M$
- $D' = D \cup \{e1, e2, s\}$

Замечание: то, что $T' = T$ следует из естественного условия на сохранение типов.

4. Удаление экземпляра ссылки с именем s между $e1$ и $e2$.

- $M' = M$
- $D' = D/\{e1, e2, s\}$

Замечание: $T' = T$ (аналогично (3)).

5. Смена значения примитивного поля.

- аналогично добавлению сслыки

4.4 Транзакции и концептуальный уровень

Для упрощения семантики работы с двунаправленными ссылками и ссылками ребёнок-родитель (являющимися частным случаем двунаправленных) естественно расширены операции удаления и добавления экземпляра обычной ссылки, являющейся частью двунаправленной:

3'. Добавление экземпляра двунаправленной ссылки с именем $s1$ между $e1$ и $e2$ (и противоположным именем $s2$).

- $M' = M$
- $D' = (D \cup \{e1, e2, s1\} \cup \{e2, e1, s2\})/X$

Здесь X — множество вытесняемых экземпляров ссылок: если мощность s_i (где $i = 1..2$) равна 0..1 или 1, то в X добавляется e_i, e_{3-i}, s_i, т. е. вытесняются экземпляры ссылок, нарушающие мощность. Замечание: то, что T' = T следует из естественного условия на сохранение типов.

4'. Удаление экземпляра двунаправленной ссылки с именем $s1$ между $e1$ и $e2$ (и противоположным именем $s2$).

- $M' = M$
- $D' = (D/\{e1, e2, s1\})/\{e2, e1, s2\}$

Дополним схему следующими ограничениями на ссылочную целостность (задаются для конкретной ссылки, неформальное описание):

1. Ограничения, выполняемые при удалении объекта на другом конце ссылки:
   - очистить ссылку
   - запретить удаление (стандартная семантика удаления объекта)
   - каскадно удалить объект в начале ссылки
2. Операции, выполняемые при удалении объекта:
   - очистить ссылку (стандартная семантика удаления)
   - каскадно удалить объект на другом конце ссылки

Данные ограничения задаются декларативно, подобно тому, как задаётся мощность ссылок. Они расширяют семантику удаления в том смысле, что операция удаления раскрывается в последовательность операций, начинающуюся с собственно удаления. Очевидно, что такая последовательность конечна, так как каждая операция удаления уменьшает количество объектов в экземпляре базы. Ограничение, запрещающее удаление непосредственно во время операции не выполняется, т. е. сама по себе такая операция всегда проходит успешно. Однако, это ограничение расширяет семантику операции *commit* (см. далее). Для различных типов ссылок существуют следующие ограничения по-умолчанию:

- для направленных ссылок — запретить удаление при удалении объекта на другом конце ссылки, очистить ссылку при удалении самого объекта.
- для агрегации — при удалении родителя каскадно удалить ребёнка, при удалении ребёнка очистить ссылку родитель-ребёнок.

Более формальное определение транзакции даётся в контексте концептуального уровня. Транзакция — преобразование экземпляра базы (M, T, D) в экземпляр (M', T, N') с той же схемой, являющееся композицией конечного количества операций. Последняя из них совпадает для всех транзакций — специальная операция commit. Замечание: операция *commit* после выполнения гарантирует выполнение всех предикатов, заданных моделью (следует из определения транзакции). Замечание: операция *rollback* не предусмотрена в терминах данного уровня — таким образом, в модели рассматриваются только успешно проведённые транзакции. На практике реализация *commit* содержит низкоуровневую транзакцию, которая может завершиться неудачно при нарушении необходимых условий, гарантируя атомарность.

Важным моментом является проверка корректности удаления ссылок и объектов. Очевидно, что результат операции по удалению ссылки или объекта (что влечёт удаление ссылки) может приводить к нарушению предикатов мощности или агрегации. Рассмотрим последовательность выполнения операции *commit*.

1. Вызов подписок на изменения модели (последовательность предварительных операций, предоставленная более высоким уровнем).
2. Проверка условий на ссылочную целостность по входящим ссылкам
3. Проверка предикатов концептуального уровня (мощности, агрегации)
4. Проверка обязательных примитивных полей
5. Проверка уникальных ключей
6. Применение операций транзакции

Проверка условий на ссылочную целостность по входящим ссылкам состоит в проверке следующего предиката: “если в исходном экземпляре существовал экземпляр ссылки $(e1, e2, s)$ и для неё запрещено удаление на другом конце, то в результирующем экземпляре либо нет ни $e1$, ни $e2$, либо есть оба этих объекта”. Предикаты мощности описаны выше.

4.5 Реализация

Для реализации данной модели были использованы средства генеративного программирования [16, 22]. Ключевая идея подхода состоит в том, что существующий абстрактный синтаксис и операционная семантика объектно-ориентированного языка программирования уже предоставляет достаточно выразительные средства, работающие с моделью памяти, аналогичной абстрактной модели и с аналогичными операциями, хотя, например, языки со сборкой мусора, в которых может отсутствовать явная операция удаления, заставляют вводить операцию удаления дополнительно, что и было сделано в силу использования Java. Таким образом, расширяя язык и, управляя генерацией кода, было предложено реализовать отображение исходной абстрактной модели, работающей в оперативной памяти, в объектную модель над базой. Полученный язык был назван DNQ (Data Navigation Query).

4.5.1 Классы и полиморфизм

Одним из наиболее логичных можно считать подход, в котором типы в базе (т. е. TY из схемы) соответствуют типам в ЯП. Каждому подтипу в иерархии ЯП соответствует отдельный тип в базе. Классы, которые должны быть сохранены в базе (хранимые) помечены (специальным модификатором, добавленным в язык). Это сделано по ряду причин: во-первых, как показывает практика, количество хранимых классов гораздо меньше, чем общее количество классов в приложении (90 из 1200 в реальном примере, использующем DNQ); во-вторых, задача сериализации в общем виде создаёт большой объём лишней работы для приложения. Вместо сериализации применён механизм, основанный на переопределении семантики указателя на экземпляр объекта (**this**) внутри реализации классов. Для его осуществления все не являющиеся статическими сигнатуры (деклараций и реализаций методов) дополняются специальным параметром, являющимся точкой входа для всех операций, ранее применимых к указателю на экземпляр. Пример: метод хранимого класса Issue с сигнатурой getUrl(User u, Boolean b), где User — хранимый класс преобразуется в метод с сигнатурой getUrl(Entity u, Boolean b, **final** Entity entity). Здесь и далее в примерах используется код на языке Java. Все вхождения **this** в зависимости от семантики преобразуются в обращения к различным сервисам, получающим на вход entity и необходимую метаинформацию (вычисленную на этапе генерации). Кроме того, во время исполнения для entity известен связанный с ней точный тип класса. Все вызовы методов объекта проходят через сервис, предоставляющий это знание. Пример: вызов issue.getUrl(u, b) преобразуется в ((IssueImpl) DnqUtils.getPersistentClassInstance(issue, "Issue")).getUrl(u, b, issue).

4.5.2 Транзакции

Вся работа с базой осуществляется в рамках транзакции. Для обеспечения гарантии отсутствия гонок (race conditions) и для снятия необходимости держать ссылку на транзакцию в коде, транзакция привязана к потоку (thread). Любые обращения к

сервисам выдают ошибку, если на момент их осуществления в текущем потоке транзакция отсутствует. Пусть на момент начала транзакции база находится в состоянии (M, T, D). Абстрактной транзакции соответствует объект в памяти, накапливающий информацию об операциях. Сервисы, через которые производится вызов операций, обладают информацией о соответствии транзакций потокам и записывают информацию о вызываемых операциях в транзакцию в соответствии с их семантикой, подробно описанной выше.

Операция commit реализована следующим образом: первые шаги (1-5) выполняются последовательно на основе записанных данных. Применение операций делегируется на более низкий уровень, в котором реализовано отображение понятий абстрактной схемы на хранилище данных с типом ключ-значение.

4.5.3 Запросы

Наиболее развитым расширением стандартного языка является язык запросов для работы с данными. Здесь мы воспользуемся понятием источника информации, введённым в LINQ — языке запросов для платформы Microsoft .NET. Идея состоит в следующем: каждый источник информации (точка входа, начиная с которой программа может начать работать с определённым подмножеством доступных данных) соответствует с точки зрения системы типов сущности, реализующей перечислимый интерфейс. Перечислимый интерфейс содержит метод, возвращающий курсор для односторонней навигации по некоторому упорядоченному множеству. В Java таким интерфейсом является Iterable<T> (введён с версии 5.0), в .NET — IEnumerable<T>, где T — тип объектов в множестве (т. е. коллекция однородная). Пример: в расширении LINQ для работы с SQL источником информации может являться таблица, синтаксис для получения таблицы Persons: **new** DataContext("params").GetTable<Person>(). Аналогично, появилась потребность в ссылке на источник информации, соответствующий множеству объектов абстрактного уровня. Так как предлагаемая реализация является in-process базой, т. е. работающей напрямую через библиотечные вызовы без удалённо-

го сервера, то контекст всегда известен, соответственно, точка входа параметризуется только типом объекта T и соответствует Iterable<T>. Для работы с такими источниками информации в Java существовал язык запросов collections language [21], интегрированный в Java, семантика которого и была взята за основу для языка запросов DNQ.

Запрос к базе — предикат, выдающий множество соответствующих ему объектов её экземпляра, либо операция, применяемая к такому множеству и выдающая упорядоченный набор его элементов. Как видно, это определение запроса применимо к стандартной объектной модели в памяти, что и было использовано при реализации collections language.

Рассмотрим систему запросов collections language:

- forEach — выполнить действие для всех элементов
- where — выборка
- select — отображение
- selectMany — отображение одного элемента в многие
- sortBy — сортировка
- intersect — пересечение
- union — объеденение
- concat — конкатенация

Эти запросы являются операциями, т. е. их синтаксис таков: input.operation(parameters), где input — источник информации (любое выражение с типом, совместимым с sequence), operation — запрос, parameters — параметры запроса (предикаты). Подобная система присутствует в Haskell и в Linq [18] и была реализована в составе проекта MPS.

5 Практическая реализация

Для практической реализации был выбран языковой инструментарий JetBrains MPS. Мы уже отметили схожесть абстрактной модели базы с моделью памяти самого ЯП. Желание работать с этими моделями единообразно и отсутствие полноценных расширений для Java, подобных LINQ стали основными предпосылками для применения генеративного подхода. Проект DNQ включает в себя язык в терминах MPS, подкреплённый библиотекой, реализующей необходимые во время исполнения сервисы.

Рассмотрим традиционный язык программирования (на примере Java). Назовём множество кода, созданное для реализации определённой задачи проектом. В традиционной схеме весь этот код во время решения задачи хранится в редактируемом виде — то есть, в текстовых файлах. Для решения задачи файлы проекта должны быть преобразованы в пригодное для целевой платформы представление — так называемый байт-код. Этим занимается компилятор, который разбирает исходный код в соответствии с его синтаксисом и строит абстрактное синтаксическое дерево (AST) [23], которое затем переводит в байт-код. Абстрактное синтаксическое дерево является концептуально чистым представлением программы, т. е. отражает те категории, которыми оперирует разработчик при создании решения задачи. Но, время жизни его в цепочке представление—AST—байт-код крайне мало и обусловлено лишь внутренними нуждами компилятора. При этом, текст является лишь одним из возможных представлений AST. Современные интегрированные среды разработки IDE дают существенное улучшение ситуации: они создают и поддерживают актуальное проекту AST на протяжении непосредственной работы над ним. Именно благодаря этому возможны средства повышения продуктивности, такие, как рефакторинги, навигация по коду и его анализ. Это уже вполне применимо при работе с одним языком. Но во всех сколько-нибудь крупных проектах как минимум, существует множество файлов содержащих метаинформацию в декларативной форме (всевозможные файлы конфигурации, скрипты для сборки и пр). Во-первых, необходимость их текстового хране-

ния привносит особенности работы с двух- или (чаще всего фактически) одномерной проекцией абстрактного дерева. Во-вторых, имея существующие компиляторы (или другие средства воплощения) для отдельных языков, невозможно полноценно связать эти языки на уровне абстрактного представления из-за того, что эти представления существуют только во время работы компиляторов, которые можно считать изолированными друг от друга.

Языковые инструментарии делают абстрактное представление основным, разделяя задачи его редактируемого представления, хранимого представления и исполнимого представления.

5.1 Синтаксис

5.1.1 Описание метаданных

Опишем синтаксис языка для работы с расширеной объектной моделью на примере хранимого класса, условно соответствующего записи (issue) в баг-трекере. Следующий листинг соответствует редактируемому представлению AST описания этого класса (то, с чем работает разработчик). В AST java присутствует понятие поля: свойства объекта в терминах вышеопианной модели декларируются как такие поля с дополнительными модификаторами. При этом модификаторы для удобства отображения хранятся в отдельном корневом узле AST (одному на пакет). Однако редактируемое представление поля даёт возможность видеть и изменять модификаторы прямо от поля, так как это делается с существующими в java модификаторами. Кроме того, централизованное хранение метаданных позволяет проще реализовать генератор: метаданные для пакета генерируются в xml-файл, загружаемый в память системой inversion of control [11] (Spring).

```
1 public persistent class Issue extends <none> implements <none> {
2   public simple string summary required;
3   public simple text description;
4   public parent Project project;
5   public directed association User[1] reporter;
```

```
6   public bidirectional association User[0..1] draftOwner
        onDelete(clear);
7   public directed association User[0..1] assignee
        onTargetDelete(clear);
8   public unordered directed association IssueLink[0..n]
        issueLinks onDelete(cascade), onTargetDelete(clear);
9 }
```

Листинг 1: Пример заголовка хранимого класса

Рассмотрим поля этого класса в контексте сценариев работы баг-трекера.

1. Поля summary и description — краткое и подробное описание issue. Как видно, информация о issue обязательно должна содержать непустое краткое описание. Работа с этими полями осуществляется с помощью стандартных операций чтения и операторов присваивания.

2. Поле Project соответсвует проекту, к которому относится данный issue. Так как это агрегация, issue должен обязательно быть привязан к какому-то проекту и при его удалении каскадно удаляется. Работа с этим полем со стороны Issue осуществляется стандартно, в то время, как противоположный конец этой ссылки в Project — поле issues является колекцией и добавление, например, нового экземпляра класса Issue происходит при помощи операции add.

3. Поле reporter — “автор описания issue”, обычная ссылка, но её мощность — 1, то есть не может быть issue без автора.

4. Поле draftOwner — “автор черновика”, двунаправленная ссылка, т. е. объект класса User (пользователь) содержит список черновиков. При удалении черновика тот автоматически удалятся из этого списка (так как использован модификатор onDelete(clear)).

5. Однонаправленная ассоциация assignee — ответственный за issue (пользователь). При удалении пользователя, ссылка на assignee очищается — issue может не иметь ответственного.

5.1.2 Манипуляция с данными

Рассмотрим дополнительно класс IssueLink, описывающий ссылку между issue (количество типов ссылок — IssueLinkType неограничено, поэтому такие ссылки не представляются метаданными как ссылки обычные).

```
1 public directed association IssueLinkType[1] linkType opts;
2 public directed association Issue[1] source
    onTargetDelete(cascade);
3 public directed association Issue[1] target
    onTargetDelete(cascade);
```

Листинг 2: Поля класса IssueLink

На его примере разберём синтаксис для операций, формально описаных в разделе 4:

1. Создание объекта *issue* с типом *Issue*: Issue issue = **new** Issue();

2. Удаление объекта *issue* с типом *Issue*: delete issue;

3. Добавление экземпляра ссылки (не множественной) с именем *reporter* между *issue* и *user*: issue.reporter = user;

   Добавление экземпляра ссылки (множественной) с именем *issueLinks* между *issue* и *issueLink*: issue.issueLinks.add(issueLink);

4. Удаление экземпляра ссылки (не множественной) с именем *reporter* между *issue* и *user*: issue.reporter = **null**;

   Удаление экземпляра ссылки (множественной) с именем *issueLinks* между *issue* и *issueLink*: issue.issueLinks.remove(issueLink);

Пусть мы хотим получить множество всех issue, связаных с данным исходящими ссылками:

```
1 public sequence<Issue> getOutgoing() {
2   return this.issueLinks2.where({~it => it.source == this;
      }).select({~it => it.target; }).distinct;
3 }
```

Листинг 3: Метод getOutgoing()

Генератор получает AST вышеприведённого запроса, проверяет, что замыкания ~it => it.source == **this** и ~it => it.target можно оптимизировать и преобразует их в соответствующий java-код.

```
1  public Iterable<Entity> getOutgoing(final Entity entity) {
2    return QueryOperations.selectDistinct(
3      new TreeKeepingEntityIterable(
4        new StaticTypedIterableDecorator(
5          "IssueLink",
6          AssociationSemantics.getToMany(entity, "issueLinks2")
7        ),
8        "IssueLink",
9        new LinkEqual("source", entity)
10     ),
11     "target"
12   );
13 }
```

Листинг 4: Метод getOutgoing() — геререированный код

Принцип работы генератора состоит в преобразовании замыканий на Java в интерпретируемые конструкции (описывающие соответствующее дерево выражений), и интерпретации этой информации во время исполнения сервисами, работающими с базой данных.

6 Заключение

В работе предложен и описан подход к работе с хранимыми данными на языке Java. В рамках подхода разработана формальная модель, дополняющая объектную модель Java и описывающая такие сущности, как хранимые объекты, ссылки, двунаправленные ссылки, агрегация, операции для удаления объектов и др., а также переопределяющая семантику работы станартных операций Java при работе с этими сущностями. Предложена модель разметки Java-программы, позволяющая встраивать декларативное описание схемы базы данных в терминах этой модели непосредственно в объявление классов хранимых объектов. В отличие от известных решений, данный подход реализует оптимизацию запросов для работы с данными не при помощи инструментирования кода на Java, а при помощи генерирования кода, снабжённого необходимой метаинформацией.

Предложенная формальная модель описывает семантику понятие транзакции и алгоритм её записи в базу. Разработана реализация этого алгоритма, реализация языкового расширения в языковом инструментарии JetBrains MPS, благодаря особенностям которого расширение является совместимым с другими возможными языковыми расширениями и имеет поддержку со стороны IDE.

Данный подход был применён при разработке многопользовательского веб-приложения с нетривиальной моделью данных и разнообразием сценариев работы с ними, что позволило выявлять недостатки как реализации языка, так и самой модели (алгоритм работы транзакции) и в конечном итоге получить удобную среду с предсказуемым поведением и легко осваиваемую неподготовленными разработчиками.

Предполагаемые направления дальнейших исследований включают разделение генератора системы запросов на независимые языковые модули, что позволит в дальнейшем предоставлять различные реализации интерпретатора для оптимизации запросов над новыми моделями данных, таких, например, как XML, с переиспользованием генератора лямбда-выражений. Также в процессе разработки находится легковесное

хранилище типа ключ-значение, которое предполагается использовать в замен существующего.

7 Приложение – пример программы на DNQ

Данный пример является переложением примера из документации Hibernate и показывает выразительность DNQ по сравнению с ним. Пример на Hibernate доступен по адресу: `http://docs.jboss.org/hibernate/stable/core/reference/en/html/example-weblog.html`.

```
public persistent class Blog {
  public simple long id sequence;
  public simple string name required;
  public ordered child BlogItem[0..n] items opts;

  public Blog() {
  }
}
```

Листинг 5: Хранимый класс Blog

```
public persistent class BlogItem {
  public simple long id sequence;
  public simple instant datetime required;
  public simple text text required;
  public simple string title required;
  public parent Blog blog opts;

  public BlogItem() {
  }
}
```

Листинг 6: Хранимый класс BlogItem

```
public class BlogMain {
  public BlogMain() {
  }

  public Blog createBlog(string name) {
    transactional {
      Blog blog = new Blog();
      blog.name = name;
      return blog;
    }
  }

  public BlogItem createBlogItem(long blogId, string title,
     string text) {
    transactional {
      BlogItem item = new BlogItem();
```

```
16        item.title = title;
17        item.text = text;
18        item.datetime = now;
19        // благодаря двунаправленности отношения родитель–ребёнок
20        // элемент item будет добавлен в коллекцию items экземпляра
              blog автоматически
21        item.blog = Blogs.where({~it => it.id == blogId; }).first;
22        return item;
23      }
24    }
25
26    public void updateBlogItem(BlogItem item, string text) {
27      transactional {
28        item.text = text;
29      }
30    }
31
32    public void updateBlogItem(long itemId, string text) {
33      transactional {
34        BlogItem blogItem = BlogItems.where({~it => it.id ==
              itemId; }).first;
35        blogItem.text = text;
36      }
37    }
38
39    public list<[long, string, int]>
          listAllBlogNamesAndItemCounts(int max) {
40      transactional {
41        returnBlogs.sortBy({~it => it.items.size; },
              asc).select({~it => [it.id, it.name, it.items.size];
              }).toList;
42      }
43    }
44
45    public Blog getBlogAndAllItems(long blogId) {
46      transactional {
47        // операция 'fetch' не нужна благодаря ленивой природе
              коллекций
48        returnBlogs.where({~it => it.id == blogId; }).first;
49      }
50    }
51
52    public list<[Blog, sequence<BlogItem>]>
          listBlogsAndRecentItems() {
53      transactional {
54        final instant minDate = now - 1 month;
```

```
55          returnBlogs.select({~blog => [blog, blog.items.where({~it
                => it.datetime > minDate; })]; }).toList;
56      }
57  }
58 }
```

Листинг 7: Примеры использования

Список литературы

[1] Hibernate community documentation, chapter 19. improving performance. `http://docs.jboss.org/hibernate/core/3.3/reference/en/html/performance.html`, 25.05.2010.

[2] Atul Adya, José A. Blakeley, Sergey Melnik, and S. Muralidhar. Anatomy of the ado.net entity framework. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 877–888, New York, NY, USA, 2007. ACM.

[3] M. M. Astrahan, Ht. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Transactions on Database Systems*, 1:97–137, 1976.

[4] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10):104–116, 1996.

[5] Christian Bauer and Gavin King. *Hibernate in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[6] Alex Buckley. A reminder of Project Lambda's scope, 2010. `http://mail.openjdk.java.net/pipermail/lambda-dev/2010-April/001299.html`.

[7] M. Darnovsky and J. Bowman. Transact-sql user's guide. *Document 3231-2.1*, 1987.

[8] Anders Hejlsberg Don Box. Linq: .net language-integrated query. 2007. `http://msdn.microsoft.com/library/bb308959.aspx`, 25.05.2010.

[9] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[10] Martin Fowler. Closure, 2004. `http://martinfowler.com/bliki/Closure.html`, 25.05.2010.

[11] Martin Fowler. Inversion of control containers and the dependency injection pattern. 2004. http://martinfowler.com/articles/injection.html, 25.05.2010.

[12] Mark Fussell. Foundations of object relational mapping. 1997.

[13] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. pages 327–336, 1991.

[14] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[15] H. V. Jagadish and Xiaolei Qian. Integrity maintenance in object-oriented databases. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*, pages 469–480, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[16] Donald E. Knuth. Literate programming. May 1984.

[17] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[18] Fabrice Marguerie, Steve Eichert, and Jim Wooley. *Linq in action*. Manning Publications Co., Greenwich, CT, USA, 2008.

[19] Jim Paterson, Stefan Edlich, Henrik Hörning, and Reidar Hörning. *The Definitive Guide to db4o*. Apress, Berkely, CA, USA, 2006.

[20] George Reese. *Database Programming with JDBC and Java, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[21] K. V. Solomatov. Collections language. 2008. http://confluence.jetbrains.net/display/MPS/Collections+language, 25.05.2010.

[22] M. Ward. Language oriented programming. 1994. http://www.dur.ac.uk/martin.ward/martin/papers/middle-out-t.pdf.

[23] Джеффри Ульман Альфред Ахо, Рави Сети. *Компиляторы: принциры, технологии, инструменты.* Вильямс, Москва, Санкт-Петербург, Киев, 2001.

[24] К. Дейт. *Введение в системы баз данных.* Диалектика, Киев, Москва, 1998.