Основные J2EE-паттерны
Data Access Object (объект доступа к данным)

Контекст

Способ доступа к данным бывает разным и зависит от источника данных. Способ доступа к персистентному хранилищу, например к базе данных, очень зависит от типа этого хранилища (реляционные базы данных, объектно-ориентированные базы данных, однородные или «плоские» файлы и т.д.) и от конкретной реализации.

Проблема

Многие реальные приложения платформы Java 2 Platform, Enterprise Edition (J2EE) должны использовать на некотором этапе персистентные данные. Для этих приложений персистентное хранение реализуется различными механизмами и существуют значительные отличия в API, используемых для доступа к этим механизмам. Другим приложениям может понадобиться доступ к данным, расположенным на разных системах. Например, данные могут находиться на мэйнфреймах, LDAP-репозиториях (Lightweight Directory Access Protocol - облегченный протокол доступа к каталогам) и т.д. Другим примером является ситуация, когда данные предоставляются службами, выполняющимися на разных внешних системах, таких как системы business-to-business (B2B), системы обслуживания кредитных карт и др.

Обычно приложения совместно используют распределенные компоненты для представления персистентных данных, например, компоненты управления данными. Считается, что приложение использует управляемую компонентом персистенцию (BMP- bean-managed persistence) для своих компонентов управления данными, если эти компоненты явно обращаются к персистентным данным - то есть компонент содержит код прямого доступа к хранилищу данных. Приложение с более простыми требованиями может вместо компонентов управления данными использовать сессионные компоненты или сервлеты с прямым доступом к хранилищу данных для извлечения и изменения данных. Также, приложение могло бы использовать компоненты управления данными с управляемой контейнером персистенцией, передавая, таким образом, контейнеру функции управления транзакциями и деталями персистенции.

Для доступа к данным, расположенным в системе управления реляционными базами данных (RDBMS), приложения могут использовать JDBC API. JDBC API предоставляет стандартный механизм доступа и управления данными в персистентном хранилище, таком как реляционная база данных. JDBC API позволяет в J2EE-приложениях использовать SQL-команды, являющиеся стандартным средством доступа к RDBMS-таблицам. Однако, даже внутри среды RDBMS фактический синтаксис и формат SQL-команд может сильно зависеть от конкретной базы данных.

Для различных типов персистентных хранилищ существует еще большее число вариантов. Механизмы доступа, поддерживаемые API и функции отличаются для различных типов персистентных хранилищ, таких как RDBMS, объектно-ориентированные базы данных, плоские файлы и т.д. Приложения, которым нужен доступ к данным, расположенным на традиционных или несовместимых системах (например, мэйнфреймы или B2B-службы), часто вынуждены использовать патентованные API. Такие источники данных представляют проблему для приложений и могут потенциально создавать прямую зависимость между кодом приложения и кодом доступа к данным. Когда бизнес-компонентам (компонентам управления данными, сессионным компонентам и даже презентационным компонентам, таким как сервлеты и вспомогательные объекты для JSP-страниц) необходим доступ к источнику данных, они могут использовать соответствующий API для получения соединения и управления этим источником данных. Но включение кода для установления соединения и доступа к данным в код этих компонентов создает тесную связь между компонентами и реализацией источника данных. Такая зависимость кода в компонентах может сделать миграцию приложения от одного типа источника данных к другому трудной и громоздкой. При изменениях источника данных компоненты необходимо изменить.

Ограничения

- Компоненты управления данными с управляемой компонентом персистенцией, сессионные компоненты, сервлеты и другие объекты, такие как вспомогательные объекты для JSP-страниц, должны получать и сохранять информацию в персистентных хранилищах и других источниках данных, например традиционных системах, B2B, LDAP и т.д..
- API доступа к персистентному хранилищу данных может зависеть от поставщика продукта. Другие источники данных могут иметь нестандартные или патентованные API. Эти API и их возможности зависят от типа хранилища данных - RDBMS, система управления объектно-ориентированными базами данных (OODBMS), XML-документы, плоские файлы и т.д. Унифицированный API доступа к этим несовместимым системам отсутствует.
- Для извлечения или сохранения данных во внешних и/или традиционных системах компоненты обычно используют патентованные API.
- Включение в компоненты специфических механизмов доступа и API прямо влияет на переносимость компонентов.
- Компоненты должны быть прозрачны для реальной реализации персистентного хранилища или источника данных и обеспечивать легкую миграцию на продукт другого поставщика, на другой тип хранилища и на другой тип источника данных.

Решение

Используйте Data Access Object (DAO) для абстрагирования и инкапсулирования доступа к источнику данных. DAO управляет соединением с источником данных для получения и записи данных.

DAO реализует необходимый для работы с источником данных механизм доступа. Источником данных может быть персистентное хранилище (например, RDBMS), внешняя служба (например, B2B-биржа), репозиторий (LDAP-база данных), или бизнес-служба, обращение к которой осуществляется при помощи протокола CORBA Internet Inter-ORB Protocol (IIOP) или низкоуровневых сокетов. Использующие DAO бизнес-компоненты работают с более простым интерфейсом, предоставляемым объектом DAO своим клиентам. DAO полностью скрывает детали реализации источника данных от клиентов. Поскольку при изменениях реализации источника данных представляемый DAO интерфейс не изменяется, этот паттерн дает возможность DAO принимать различные схемы хранилищ без влияния на клиенты или бизнес-компоненты. По существу, DAO выполняет функцию адаптера между компонентом и источником данных.

Структура

На рисунке 9.1 показана диаграмма классов, представляющая взаимоотношения в паттерне DAO.

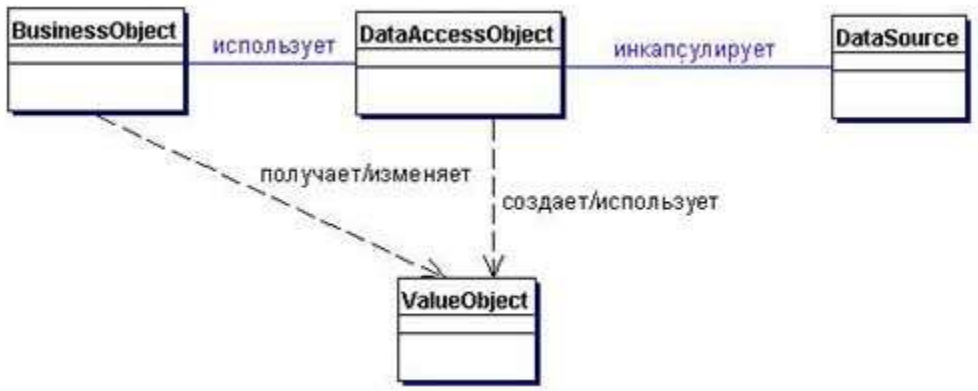

Рисунок 9.1 Data Access Object

Участники и обязанности

На рисунке 9.2 представлена диаграмма последовательности действий, показывающая взаимодействия между различными участниками в данном паттерне.

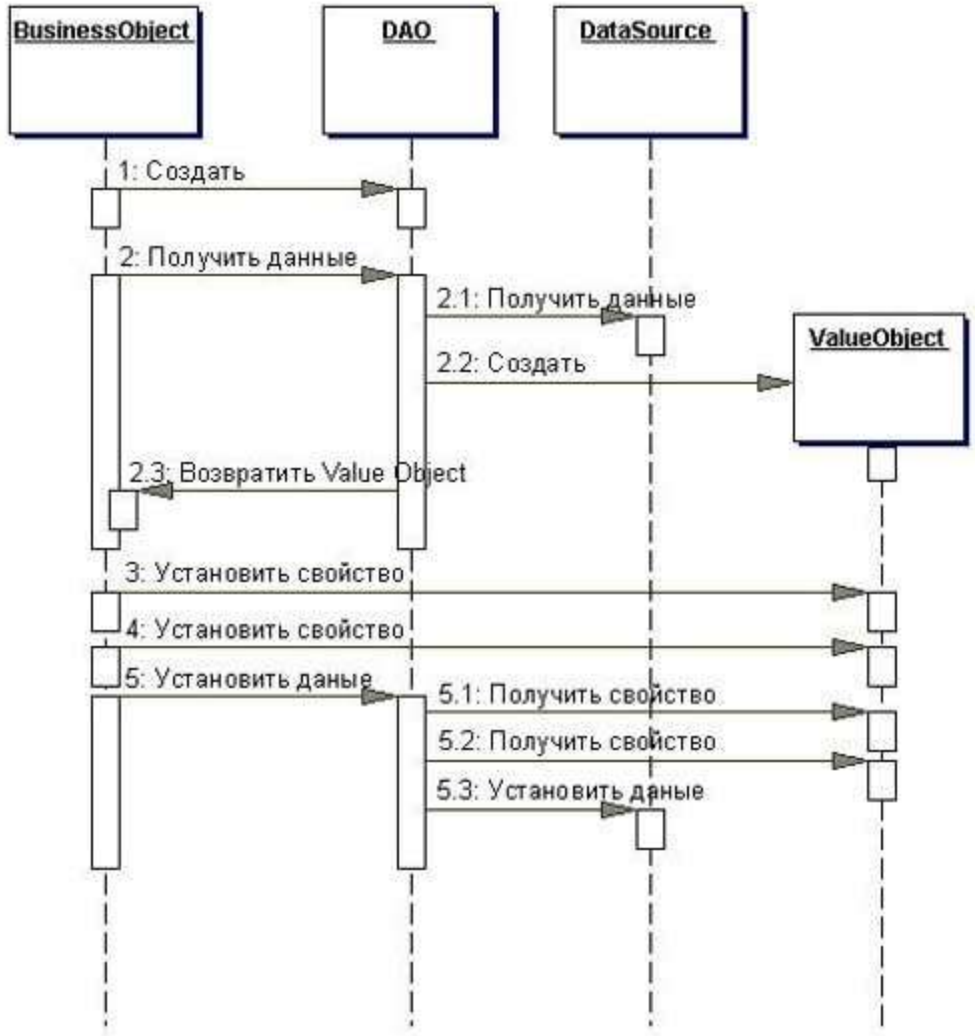

Рисунок 9.2 Диаграмма последовательности действий паттерна Data Access Object

BusinessObject

BusinessObject представляет клиента данных. Это объект, который нуждается в доступе к источнику данных для получения и сохранения данных. BusinessObject может быть реализован как сессионный компонент, компонент управления данными или другой Java-объект, сервлет или вспомогательный компонент.

DataAccessObject

DataAccessObject является первичным объектом данного паттерна. DataAccessObject абстрагирует используемую реализацию доступа к данным для BusinessObject, обеспечивая прозрачный доступ к источнику данных. BusinessObject передает также ответственность за выполнение операций загрузки и сохранения данных объекту DataAccessObject.

DataSource

Представляет реализацию источника данных. Источником данных может быть база данных, например, RDBMS, OODBMS, XML-репозиторий, система плоских файлов и др. Источником данных может быть также другая система (традиционная/мэйнфрейм), служба (B2B-служба или система обслуживания кредитных карт), или какой-либо репозиторий (LDAP).

TransferObject

Представляет собой Transfer Object, используемый для передачи данных. DataAccessObject может использовать Transfer Object для возврата данных клиенту. DataAccessObject может также принимать данные от клиента в объекте Transfer Object для их обновления в источнике данных.

Стратегии

Стратегия Automatic DAO Code Generation

Поскольку BusinessObject соответствует конкретному DAO, есть возможность установить взаимоотношения между BusinessObject, DAO, и применяемыми реализациями (таблицы в RDBMS). После установления взаимоотношений появляется возможность написать простую утилиту для генерации кода, зависящую от приложения, которая может генерировать код для всех нужных приложению объектов DAO. Метаданные для генерации DAO могут определяться разработчиком в файле-дескрипторе. В качестве альтернативы генератор кода может автоматически проанализировать базу данных и предоставить необходимые для доступа к ней объекты DAO. Если требования к DAO являются достаточно сложными, можно использовать интсрументальные средства сторонних производителей, обеспечивающие отображение «объектный-реляционный» для баз данных RDBMS. Такие средства обычно имеют GUI-интерфейс для отображения бизнес-объектов в объекты персистентного хранилища и определяют промежуточные объекты DAO. Инструментальные средства автоматически генерируют код после завершения отображения и могут предоставлять другие ценные функции, например, кэширование результатов, кэширование запросов, интеграция с серверами приложений, интеграция с другими сторонними продуктами (например, распределенное кэширование) и т.д.

Стратегия Factory for Data Access Objects

Паттерн DAO может быть сделан очень гибким при использовании паттернов Abstract Factory [GoF] и Factory Method [GoF] (см. секцию "Связанные паттерны" в этом разделе).

Данная стратегия может быть реализована с использованием паттерна Factory Method для генерации нескольких объектов DAO, которые нужны приложению, в тех случаях, когда применяемое хранилище данных не изменяется при переходе от одной реализации к другой. Диаграмма классов этого случая приведена на рисунке 9.3.

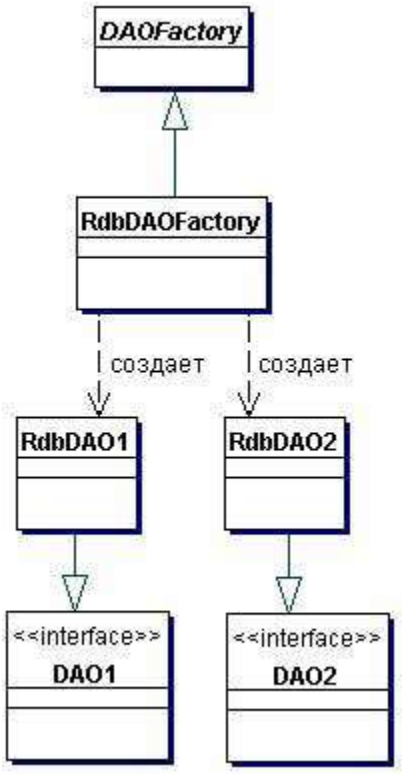

Рисунок 9.3 Стратегия Factory for Data Access Object, использующая Factory Method

Когда используемое хранилище данных может измениться при переходе от одной реализации к другой, данная стратегия может быть реализована с применением паттерна Abstract Factory. Abstract Factory, в свою очередь, может создать и использовать реализацию Factory Method implementation, как рекомендуется в книге *«Паттерны проектирования: Элементы повторно используемого объектно-ориентированного программного обеспечения» [GoF]*. В этом случае данная стратегия предоставляет абстрактный объект генератора DAO (Abstract Factory), который может создавать конкретные генераторы DAO различного типа, причем каждый генератор может поддерживать различные типы реализаций персистентных хранилищ данных. После получения конкретного генератора для конкретной реализации вы можете использовать его для генерации объектов DAO, поддерживаемых и реализуемых в этой реализации.

Диаграмма классов этой стратегии представлена на рисунке 9.4. Эта диаграмма классов показывает базовый генератор DAO, являющийся абстрактным классом, который наследуется и реализуется различными конкретными генераторами DAO для поддержки доступа к специфической реализации хранилища данных. Клиент может получить реализацию конкретного генератора DAO, например RdbDAOFactory, и использовать его для получения конкретных объектов DAO, работающих с этой конкретной реализацией хранилища данных. Например, клиент

может получить RdbDAOFactory и использовать его для получения конкретных DAO, таких как RdbCustomerDAO, RdbAccountDAO и др. Объекты DAO могут расширять и реализовывать общий базовый класс (показанные как DAO1 и DAO2) и детально описывать требования к DAO для поддерживаемых бизнес-объектов. Каждый конкретный объект DAO отвечает за соединение с источником данных и за получение и управление данными для поддерживаемого им бизнес-объекта.

Пример реализации паттерна DAO и его стратегий приведен в секции «Пример» данного раздела.

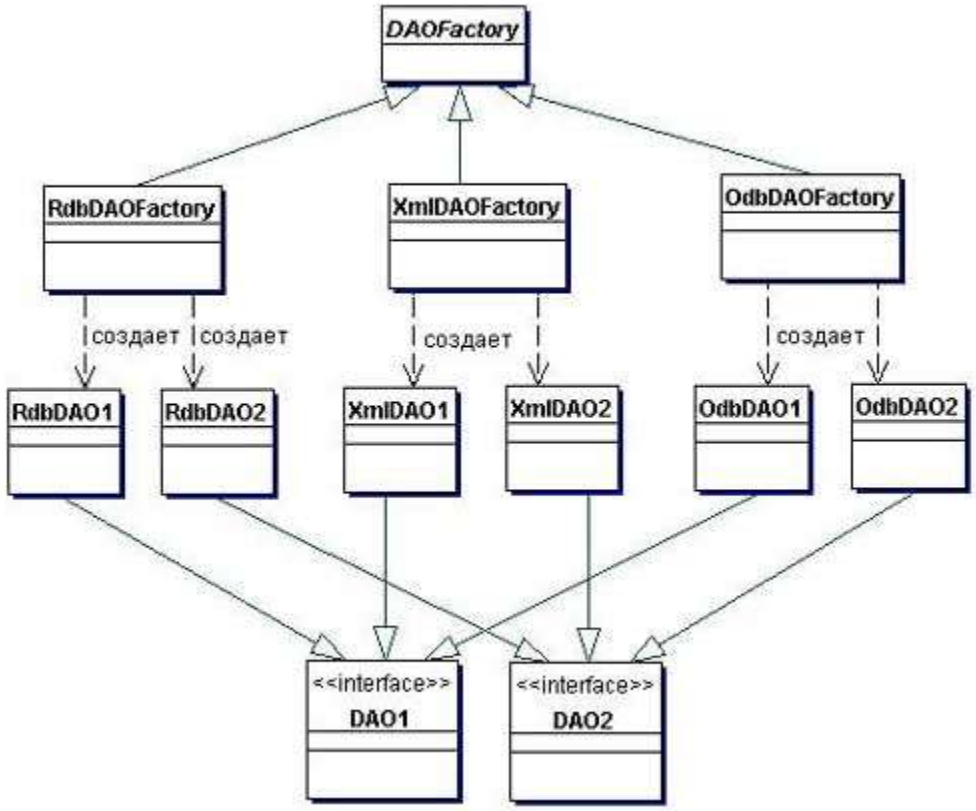

Рисунок 9.4 Стратегия Factory for Data Access Object, использующая Abstract Factory

Диаграмма последовательности действий, описывающая взаимодействия для этой стратегии, представлена на рисунке 9.5.

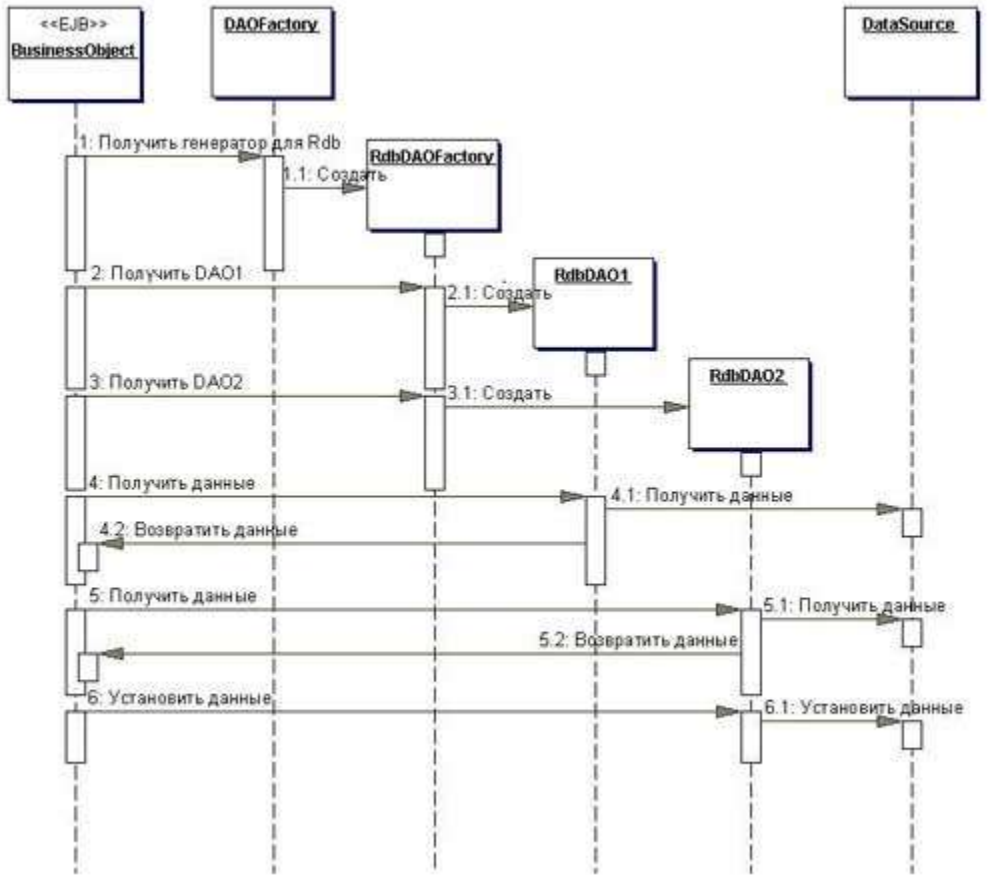

Рисунок 9.5 Диаграмма последовательности действий для стратегии Factory for Data Access Objects, использующей Abstract Factory

Выводы

- **Разрешает прозрачность**
  Бизнес-объекты могут использовать источник данных, не имея знаний о конкретных деталях его реализации. Доступ является прозрачным, поскольку детали реализации скрыты внутри DAO.
- **Облегчает миграцию**
  Уровень объектов DAO облегчает приложению миграцию на другую реализацию базы данных. Бизнес-объекты не знают о деталях реализации используемых данных. Следовательно, процесс миграции требует изменений только в уровне DAO. Более того, при использовании стратегии генератора можно предоставить конкретную реализацию генератора для каждой реализации хранилища данных. В этом случае миграция на другую реализацию хранилища означает предоставление приложению новой реализации генератора.
- **Уменьшает сложность кода в бизнес-объектах**
  Поскольку объекты DAO управляют всеми сложностями доступа к данным, упрощается код бизнес-компонентов и других клиентов данных, использующих DAO. Весь зависящий от реализации код (например, SQL-команды) содержится в DAO, а не в бизнес-объекте. Это улучшает читаемость кода и производительность разработки.

- **Централизует весь доступ к данным в отдельном уровне**
  Поскольку все операции доступа к данным реализованы в объектах DAO, отдельный уровень доступа к данным может рассматриваться как уровень, изолирующий остальную часть приложения от реализации доступа к данным. Такая централизация облегчает поддержку и управление приложением.
- **Бесполезна для управляемой контейнером персистенции**
  Поскольку EJB-контейнер управляет компонентами с управляемой контейнером персистенцией (CMP - container-managed persistence), контейнер автоматически обслуживает весь доступ к хранилищу данных. Приложения, использующие компоненты управления данными этого типа, не нуждаются в уровне объектов DAO, поскольку сервер приложений обеспечивает эту функциональность. Однако, объекты DAO остаются полезны в случаях, когда необходимо использовать комбинацию CMP (для компонентов управления данными) и BMP (для сессионных компонентов, сервлетов).
- **Добавляет дополнительный уровень**
  Объекты DAO создают дополнительный уровень объектов между клиентом данных и источником данных, который должен быть разработан и реализован для использования преимуществ, предлагаемых данным паттерном. Но за реализуемые при этом преимущества приходится платить дополнительными усилиями при разработке.
- **Требуется разработка иерархии классов**
  При использовании стратегии генератора необходимо разработать и реализовать иерархию конкретных генераторов и иерархию конкретных объектов, производимых генераторами. Эти дополнительные усилия необходимо принимать во внимание, если существует достаточно оснований для реализации такой гибкости. Это увеличивает сложность разработки. Однако, вы можете сначала реализовать эту стратегию с паттерном Factory Method, а затем, при необходимости, перейти к паттерну Abstract Factory.

Пример

Реализация паттерна Data Access Object

Код объекта DAO для персистентного объекта, предоставляющего информацию о клиенте (Customer), представлен в примере 9.4. CloudscapeCustomerDAO создает объект Customer Transfer Object при вызове метода findCustomer().

Код, использующий DAO, показан в примере 9.6. Диаграмма классов этого примера приведена на рисунке 9.6.

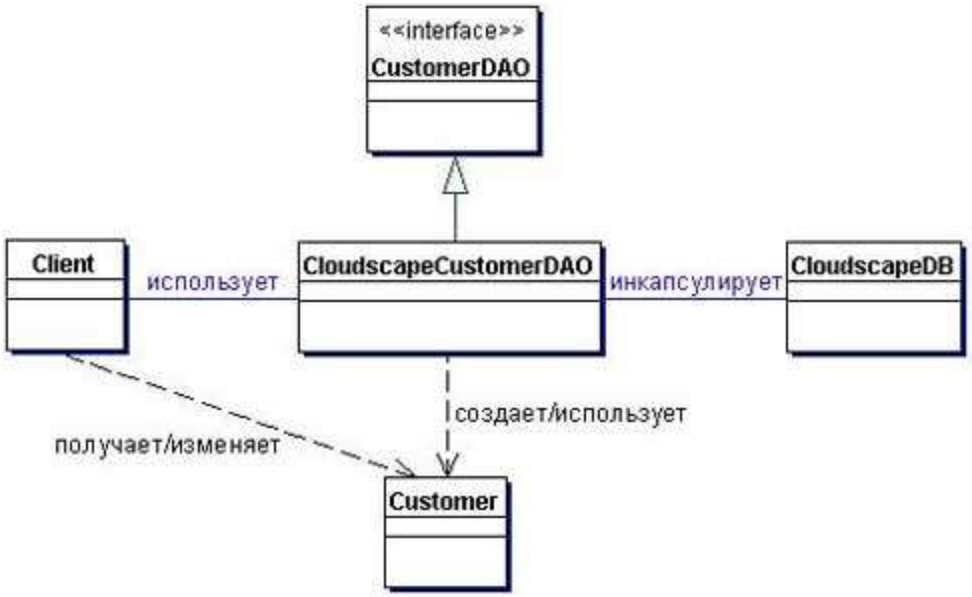

Рисунок 9.6 Реализация паттерна DAO

Реализация стратегии Factory for Data Access Objects

Использование паттерна Factory Method

Рассмотрим пример, в котором мы применяем данную стратегию для создания генератором многих объектов DAO для одной реализации базы данных (например, Oracle). Генератор производит такие объекты DAO, как CustomerDAO, AccountDAO, OrderDAO и др. Диаграмма классов для этого примера приведена на рисунке 9.7.

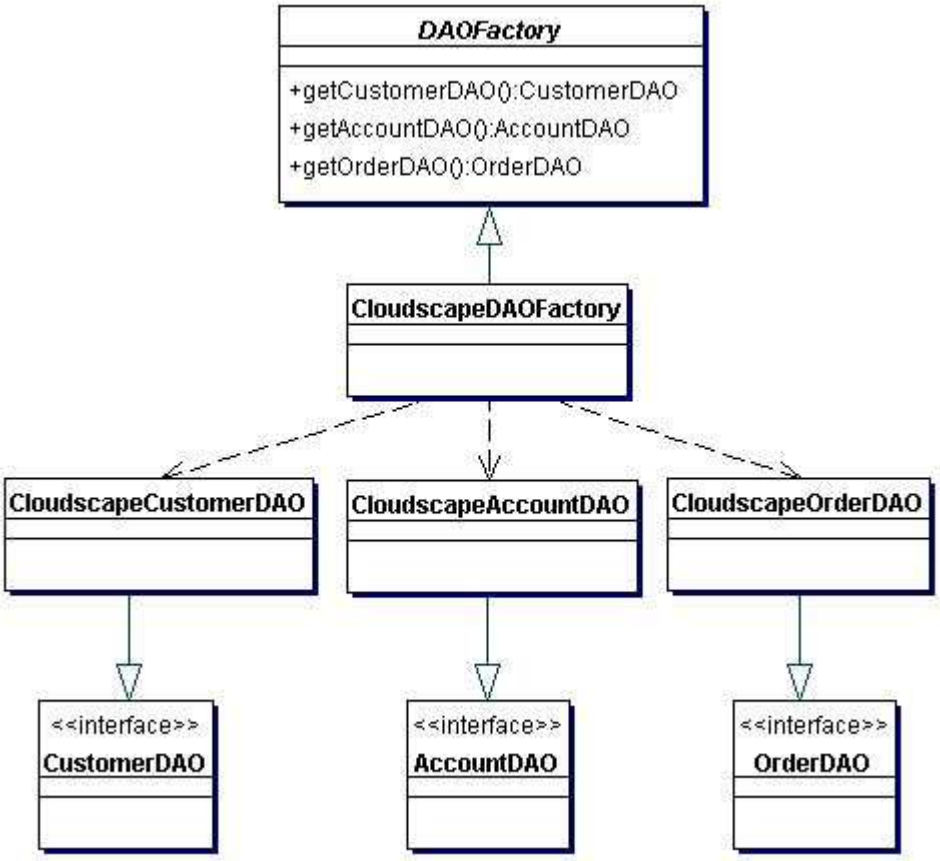

Рисунок 9.7 Реализация стратегии Factory for DAO, использующей Factory Method

Код генератора DAO (CloudscapeDAOFactory) приведен в примере 9.2.

Использование паттерна Abstract Factory

Рассмотрим пример, в котором мы применяем данную стратегию для трех различных баз данных. В этом случае может быть использован паттерн Abstract Factory. Диаграмма классов этого примера показана на рисунке 9.8. Код в примере 9.1 показывает фрагмент абстрактного класса DAOFactory. Этот генератор производит такие объекты DAO как CustomerDAO, AccountDAO, OrderDAO и др. Данная стратегия использует реализацию Factory Method в генераторах, созданных при помощи Abstract Factory.

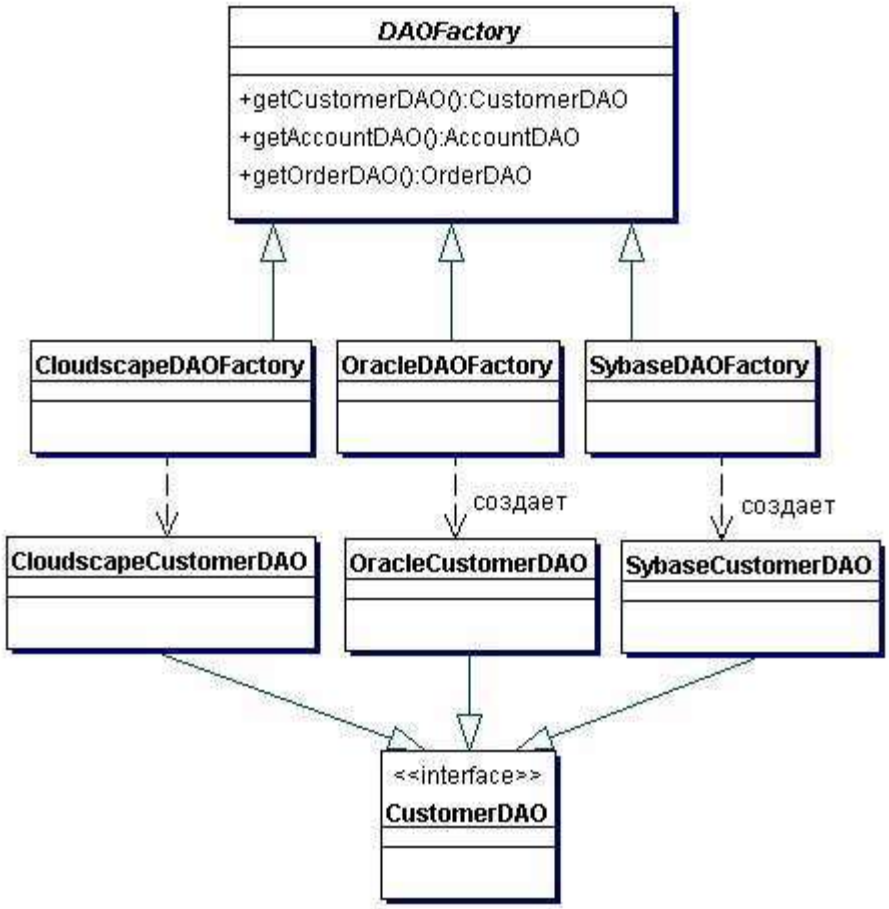

Рисунок 9.8 Реализация стратегии Factory for DAO, использующей Abstract Factory

Пример 9.1 Абстрактный класс DAOFactory

```
// Абстрактный класс DAO Factory
public abstract class DAOFactory {

 // Список типов DAO, поддерживаемых генератором
  public static final int CLOUDSCAPE = 1;
  public static final int ORACLE = 2;
  public static final int SYBASE = 3;
  ...

 // Здесь будет метод для каждого DAO, который может быть
  // создан. Реализовывать эти методы
  // должны конкретные генераторы.
 public abstract CustomerDAO getCustomerDAO();
  public abstract AccountDAO getAccountDAO();
  public abstract OrderDAO getOrderDAO();
  ...

  public static DAOFactory getDAOFactory(
      int whichFactory) {

    switch (whichFactory) {
```

```
      case CLOUDSCAPE:
          return new CloudscapeDAOFactory();
      case ORACLE    :
          return new OracleDAOFactory();
      case SYBASE    :
          return new SybaseDAOFactory();
      ...
      default           :
          return null;
    }
  }
}
```

Код CloudscapeDAOFactory приведен в примере 9.2. Реализации OracleDAOFactory и SybaseDAOFactory аналогичны, за исключением особенностей каждой реализации, таких как JDBC-драйвер, URL базы данных и различий в синтаксисе SQL, если таковые имеются.

Пример 9.2 Конкретная реализация DAOFactory для Cloudscape

```
// Конкретная реализация DAOFactory  для Cloudscape
import java.sql.*;

public class CloudscapeDAOFactory extends DAOFactory {
  public static final String DRIVER=
    "COM.cloudscape.core.RmiJdbcDriver";
  public static final String DBURL=
    "jdbc:cloudscape:rmi://localhost:1099/CoreJ2EEDB";

 // метод для создания соединений к Cloudscape
  public static Connection createConnection() {
  // Использовать DRIVER и DBURL для создания соединения
    // Рекомендовать реализацию/использование пула соединений
 }
  public CustomerDAO getCustomerDAO() {
  // CloudscapeCustomerDAO реализует CustomerDAO
    return new CloudscapeCustomerDAO();
  }
  public AccountDAO getAccountDAO() {
  // CloudscapeAccountDAO реализует AccountDAO
    return new CloudscapeAccountDAO();
  }
  public OrderDAO getOrderDAO() {
  // CloudscapeOrderDAO реализует OrderDAO
    return new CloudscapeOrderDAO();
  }
  ...
}
```

Интерфейс CustomerDAO, показанный в примере 9.3, определяет DAO-методы для персистентного объекта Customer, которые реализованы всеми конкретными реализациями DAO, такими как, CloudscapeCustomerDAO, OracleCustomerDAO и SybaseCustomerDAO. Аналогично (но здесь не приводится) реализуются интерфейсы AccountDAO и OrderDAO, определяющие DAO-методы соответственно для бизнес-объектов Account и Order.

Пример 9.3 Базовый DAO-интерфейс для Customer

```
// Интерфейс, который должны поддерживать все CustomerDAO
public interface CustomerDAO {
  public int insertCustomer(...);
  public boolean deleteCustomer(...);
  public Customer findCustomer(...);
  public boolean updateCustomer(...);
  public RowSet selectCustomersRS(...);
  public Collection selectCustomersTO(...);
  ...
}
```

CloudscapeCustomerDAO реализует CustomerDAO, как показаго в примере 9.4. Реализации других объектов DAO, таких как CloudscapeAccountDAO, CloudscapeOrderDAO, OracleCustomerDAO, OracleAccountDAO, и т.д. - аналогичны.

Пример 9.4 Реализация Cloudscape DAO для Customer

```
// Реализация CloudscapeCustomerDAO
// интерфейса CustomerDAO. Этот класс может содержать весь
// специфичный для Cloudscape код и SQL-команды.
// Клиент, таким образом, защищается от необходимости
// знать эти детали реализации.

import java.sql.*;

public class CloudscapeCustomerDAO implements
    CustomerDAO {

  public CloudscapeCustomerDAO() {
  // инициализация
  }

  // Следующие методы по необходимости могут использовать
 // CloudscapeDAOFactory.createConnection()
 // для получения соединения

  public int insertCustomer(...) {
    // Реализовать здесь операцию добавления клиента.
  // Возвратить номер созданного клиента
  // или -1 при ошибке
  }

  public boolean deleteCustomer(...) {
    // Реализовать здесь операцию удаления клиента.
  // Возвратить true при успешном выполнении, false при ошибке
 }

  public Customer findCustomer(...) {
    // Реализовать здесь операцию поиска клиента, используя
    // предоставленные значения аргументов в качестве критерия поиска.
  // Возвратить объект Transfer Object при успешном поиске,
  // null или ошибку, если клиент не найден.
 }
```

```
  public boolean updateCustomer(...) {
    // Реализовать здесь операцию обновления записи,
  // используя данные из customerData Transfer Object
  // Возвратить true при успешном выполнении, false при ошибке
  }

  public RowSet selectCustomersRS(...) {
    // Реализовать здесь операцию выбора клиентов,
    // используя предоставленный критерий.
  // Возвратить RowSet.
 }

  public Collection selectCustomersTO(...) {
    // Реализовать здесь операцию выбора клиентов,
  // используя предоставленный критерий.
    // В качестве альтернативы, реализовать возврат
  // коллекции объектов Transfer Object.
  }
  ...
}
```

Класс Customer Transfer Object показан в примере 9.5. Он используется объектами DAO для передачи и приема данных от клиентов. Использование объектов Transfer Object детально рассматривалось в паттерне Transfer Object.

Пример 9.5 Customer Transfer Object

```
public class Customer implements java.io.Serializable {
 // переменные-члены
  int CustomerNumber;
  String name;
  String streetAddress;
  String city;
  ...

 // методы getter и setter...
  ...
}
```

В примере 9.6 показано использование генератора DAO и DAO. Если реализация меняется от Cloudscape к другому продукту, необходимо изменить только вызов метода getDAOFactory() в генераторе DAO для получения другого генератора.

Пример 9.6 Использование DAO и DAO-генератора - код клиента

```
...
// создать требуемый генератор DAO
DAOFactory cloudscapeFactory =
  DAOFactory.getDAOFactory(DAOFactory.DAOCLOUDSCAPE);

// Создать DAO
CustomerDAO custDAO =
  cloudscapeFactory.getCustomerDAO();

// создать нового клиента
```

```
int newCustNo = custDAO.insertCustomer(...);

// найти объект customer. Получить объект Transfer Object.
Customer cust = custDAO.findCustomer(...);

// изменить значения в Transfer Object.
cust.setAddress(...);
cust.setEmail(...);
// обновить объект customer, используя DAO
custDAO.updateCustomer(cust);

// удалить объект customer
custDAO.deleteCustomer(...);
// выбрать всех клиентов одного города
Customer criteria=new Customer();
criteria.setCity("New York");
Collection customersList =
  custDAO.selectCustomersTO(criteria);
// возвратить customersList - коллекцию объектов Customer
// Transfer Objects. Проходит по коллекции для
// получения значений.

...
```

Связанные паттерны

- **Transfer Object**
  DAO использует Transfer Objects для передачи данных клиентам и от них.
- **Factory Method [GoF] и Abstract Factory [GoF]**
  Стратегия *Factory for Data Access Objects* использует паттерн Factory Method для реализации конкретных генераторов и их продуктов (объектов DAO). Для дополнительной гибкости, в стратегиях может быть применен паттерн Abstract Factory, как рассматривалось выше.
- **Broker [POSA1]**
  Паттерн DAO связан с паттерном Broker, который описывает подходы для разделения клиентов и серверов в распределенных системах. Паттерн DAO применяет этот паттерн более конкретно для разделения уровня ресурсов от клиентов и перемещения его в другой уровень, такой как бизнес-уровень, или уровень представлений.