Проектирование ПО

Тема 8. Структурный рефакторинг

Рефакторинг

Рефакторинг (**refactoring**) — процесс улучшения внутренней структуры ПО без изменения внешнего поведения кода.

Хорошо сочетается с быстрой разработкой. Он может проводиться в любом месте итерации, но наиболее эффективно выполнять его либо в конце текущей итерации, либо в начале следующей.

Время и трудозатраты на сопровождение кода существенно превышают время и трудозатраты на написание кода. При сопровождении кода необходимо прочитать и попробовать понять его, чтобы затем изменить или расширить. Рефакторинг кода облегчает процесс сопровождения.

Цели рефакторинга

Целью рефакторинга является устранение следующих нежелательных свойств кода:

- **дублированный код** — одни и те же части кода в нескольких местах;
- **длинная подпрограмма-метод** — метод, который делает слишком много;
- **большой класс** — класс, который делает слишком много и/или имеет слишком много элементов данных;
- **длинный список параметров** — слишком большое количество данных передается в качестве параметров;
- **расширяющееся изменение** — когда класс должен быть изменен в результате больше чем одного вида изменения;
- **эффект дробовика** — когда одно изменение воздействует на несколько классов;
- **излишняя зависимость** — метод, который обращается ко многим другим объектам с помощью сообщений get (получить), чтобы получить данные для собственных вычислений;
- **группы данных** — данные, которые обычно используются вместе во многих местах и которые следует преобразовать в объект.

Методы рефакторинга

Методы рефакторинга — основные принципы и лучшая практика изменения кода с целью улучшения его понятности, удобства сопровождения и масштабируемости.

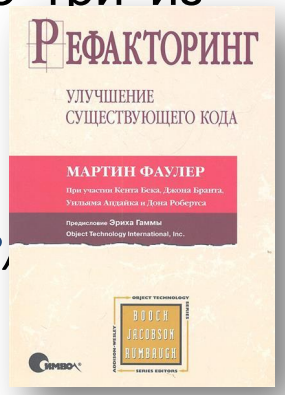
Небольшие изменения кода могут привести к впечатляющим результатам. К сожалению, можно также и ухудшить код.

CASE-средства могут существенно помочь в реализации рефакторинга. Многие инструментальные средства содержат каталоги поддерживаемых рефакторингов.

Фаулер перечисляет более шестидесяти методов рефакторинга. Рассмотрим только три из них:

- ***Класс извлечения*** ***(Extract Class)***;
- ***Метод подключения*** ***(Subsume Method)***;
- ***Интерфейс извлечения*** ***(Extract Interface)***.

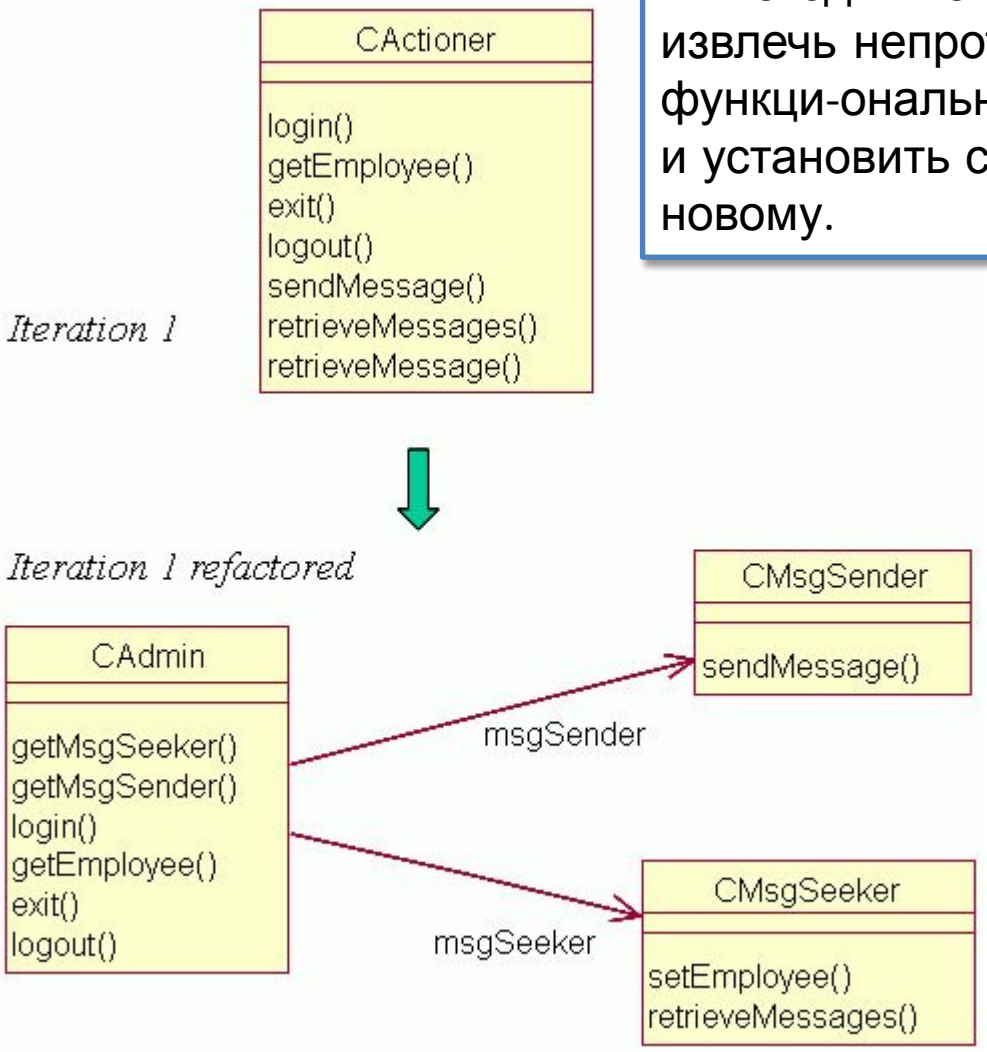
Мартин Фаулер (Martin Fowler)



Класс извлечения *(Extract Class)*

В случае большого класса уместны: *Класс извлечения* и *Интерфейс извлечения.*

Рефакторинг ***Класс извлечения***: «Создайте новый класс и переместите соответствующие поля и методы из старого класса в новый». Нужно извлечь непротиворечивые и объединенные части функци-ональных возможностей в отдельный класс и установить связь ассоциации от старого класса к новому.



Из класса CActioner выделены методы извлечения исходящих сообще-ний (класс CMsgSeeker) и посылки сообщений (CMsgSender).

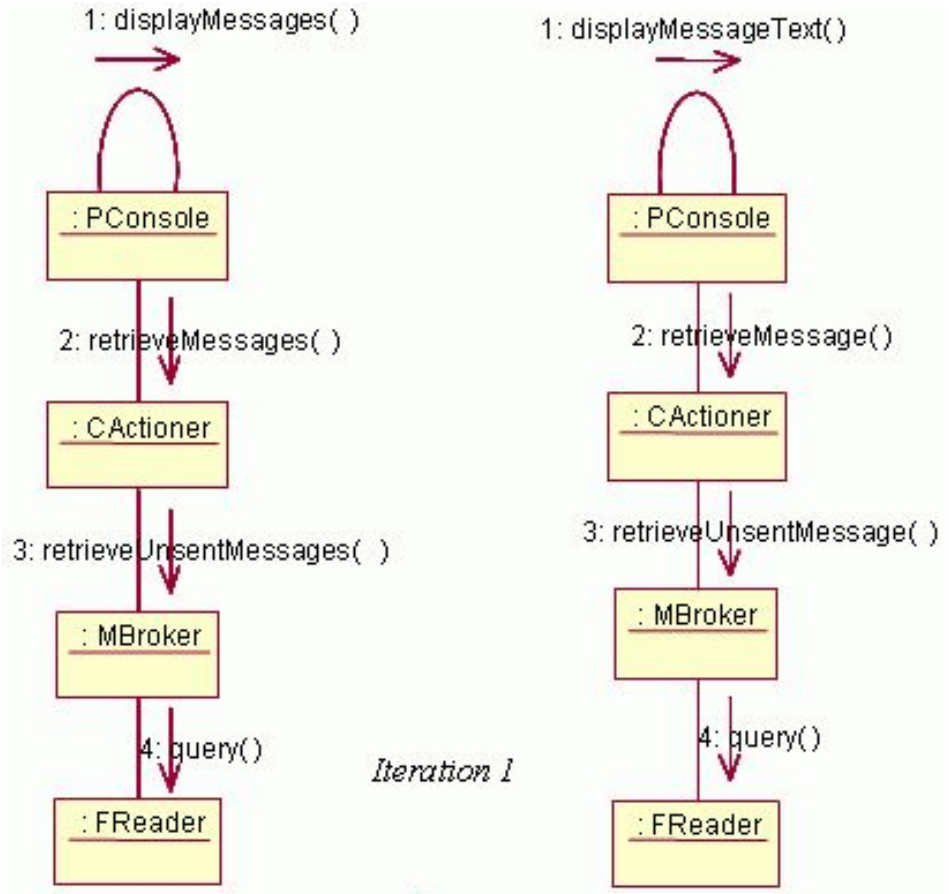
Связи поддерживаются двумя мето-дами: getMsgSeeker() и GetMsgSender(). Первый получает объ ект CMsgSeeker, второй – CMsgSender.

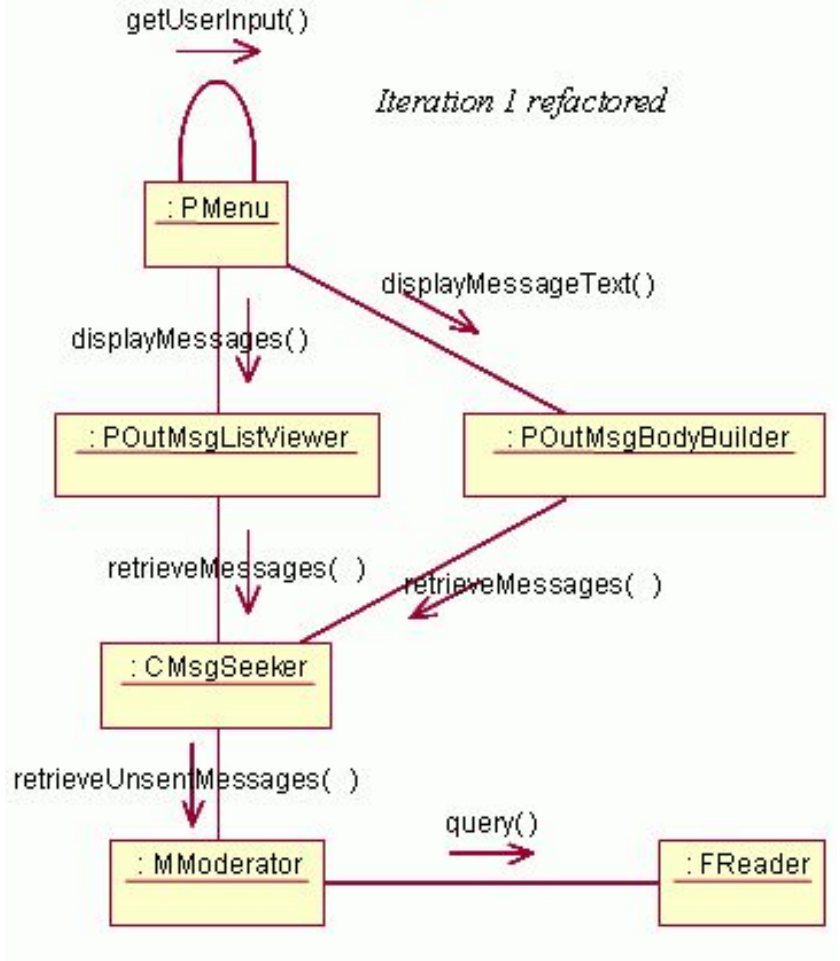
setEmployee() используется, чтобы установить ассоциацию от CMsgSeeker к EEmployee. setEmployee() получает объект EEmployee от метода login().

Метод подключения *(Subsume Method)*

Метод подключения устраняет метод включением его функциональных возможностей в другой существующий метод.

Метод подключения устраняет метод retrieveMessage() включением его возможностей в метод retrieveMessages().

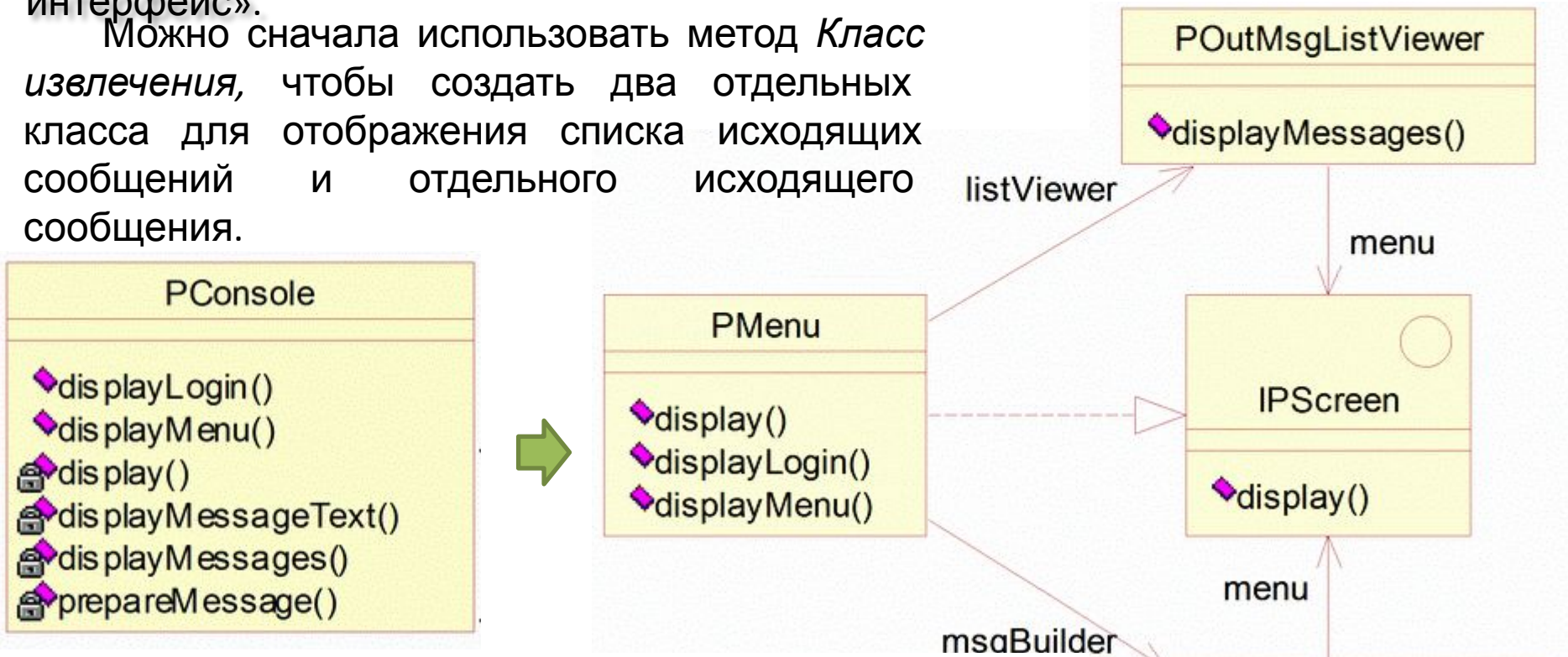




Интерфейс извлечения

Интерфейс извлечения: «Несколько клиентов используют то же самое подмножество интерфейса класса или два класса содержат общую часть своих интерфейсов». Метод используется, чтобы «выделить подмножество в интерфейс».

Можно сначала использовать метод *Класс извлечения,* чтобы создать два отдельных класса для отображения списка исходящих сообщений и отдельного исходящего сообщения.



Два новых класса уровня используют один и тот же метод display(), чтобы показать строку на экране. Будет лучше, если метод display() будет извлечен в интерфейс, реализован в PMenu и использован при необходимости новыми классами.

Паттерны рефакторинга

Паттерны рефакторинга — структурные паттерны, используемые в рефакторинге кода.

Мартин Фаулер называет их паттернами структуры промышленного приложения. Они являются целью тех аспектов системы, которые делают ее промышленным приложением в противоположность маломасштабным настольным приложениям. Они имеют дело с такими проблемами, как связь с БД, кэши сохраняемых объектов, расположенные в памяти, управление транзакциями и параллелизмом, представление Web-технологий, работа с распределенными объектами и т. д.

Книга Фаулера перечисляет и документирует более пятидесяти структурных паттернов.

Паттерны представлены следующими группами:

- ***Коллекция идентичности объектов*** (**Identity Ma**
- ***Преобразователь данных*** (**Data Mapper**);
- ***Загрузка по требованию*** (**Lazy Load**);
- ***Единица работы*** (**Unit of Work**).

Коллекция идентичности объектов (Identity Map)

Паттерн ***Коллекция идентичности объектов*** «гарантирует, что каждый объект загружается только однажды, сохраняя ссылку на объект в коллекции. При использовании объектов просматривается коллекция, содержащая ссылки на них»

Фаулер различает ***явно заданные*** (***explicit***) и ***общие*** (***generic***) коллекции.

К объектам в *явно заданной коллекции идентичности объектов* можно иметь доступ (get), их можно зарегистрировать (put) и снять с них регистрацию (remove) с использованием *различных* методов для каждого класса кэшированных объектов. Например, для получения доступа к объекту: getEEmployee (new Integer(empOID));

Для объектов в *общей коллекции* каждая из этих операций выполняются единственным методом для всех классов. Например: get ("EEmployee", new Integer(OID)).

Фаулер различает ***одну коллекцию на класс*** и ***одну коллекцию на сеанс***.

Одна коллекция на сеанс требует глобальных уникальных идентификаторов объектов. *Одна коллекция на класс* может использовать идентификаторы объектов, уникальные в пределах класса.

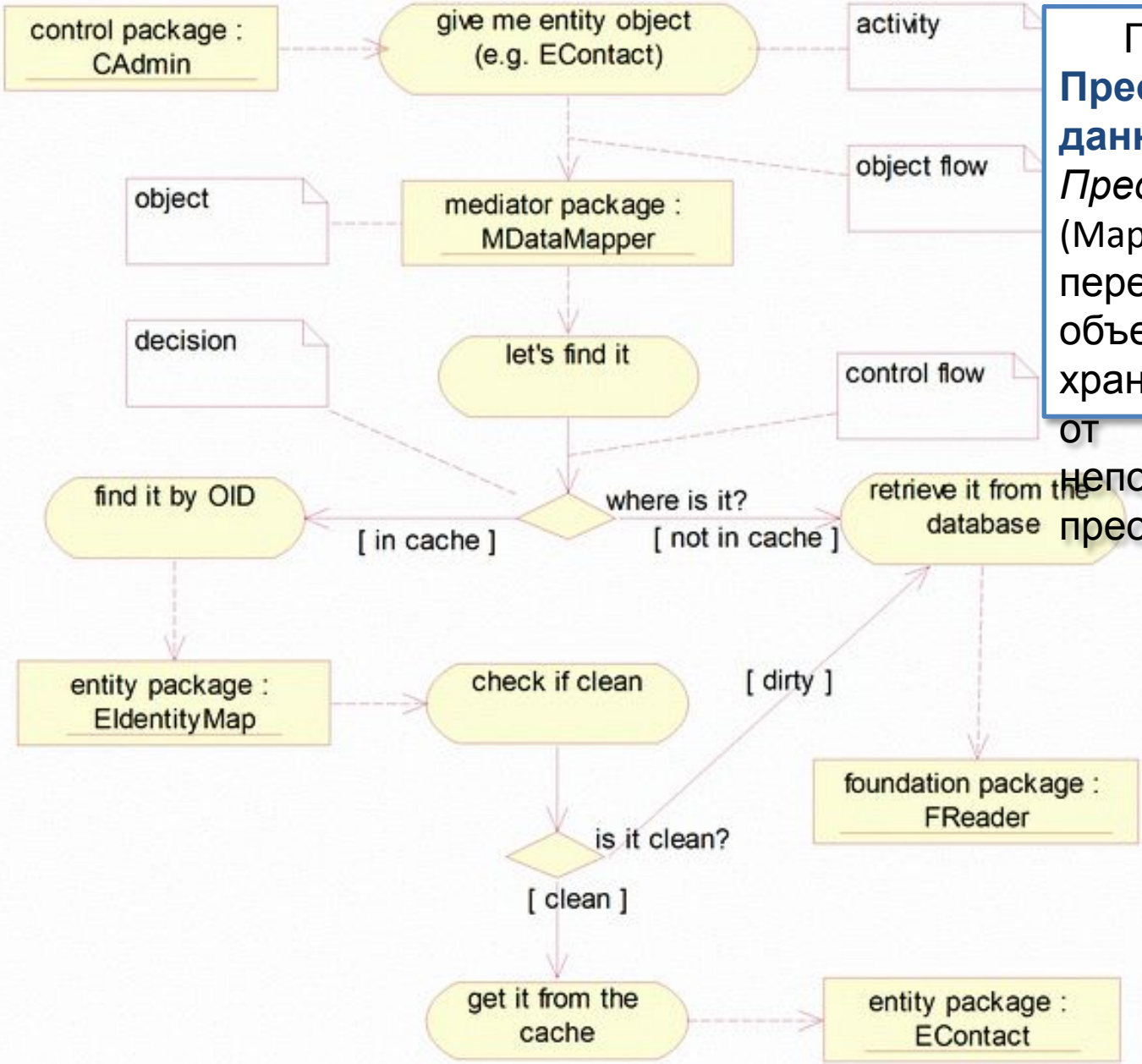
В обоих случаях для коллекций идентичности обновляемых объектов должна быть обеспечена защита в виде транзакций и они должны быть помечены как измененные, когда объекты идентичности выходят из

Коллекция идентичности объектов

EIdentityMap — ***явно заданная коллекция идентичности объектов*** - она имеет отдельные методы для каждого класса пакета entity. Класс представляет стратегию ***одной коллекции на сеанс*** — он использует глобальные уникальные идентификаторы OID.

```
18: public class EIdentityMap {
21:   private Map OIDToObj;   //OID -> Obj
22:   private Map msgPKToOID;     //msgPK -> OID
23:
24:   public EIdentityMap() {
25:     OIDToObj = new HashMap();
28:     msgPKToOID = new HashMap();
29:   }
30:
31:   /** Получить хранимого делового партнера */
32:   public IAContact findContact(int contactOID) {
33:     return (IAContact) OIDToObj.get(new Integer(contactOID));
34:   }
35:   /** Разместить делового партнера с заданным OID */
36:   public void registerContact(IEObjectlD oidObject) {
37:     OIDToObj.put(new Integer(oidObject.getOID()), oidObject);
41:   }
42:   /** Снять регистрацию с зарегистрированного делового партнера */
43:   public void unregisterContact(IEObjectlD oidContact) {
44:     OIDToObj.remove(new Integer (oidContact.getOID()));
46:   }
113:}
```

Преобразователь данных (Data Mapper)



Паттерн **Преобразователь данных**: «слой *Преобразова-телей* (Mappers), который перемещает данные между объектами и БД, когда они хранятся независимо друг от друга, и непосредственно сам преобразователь».

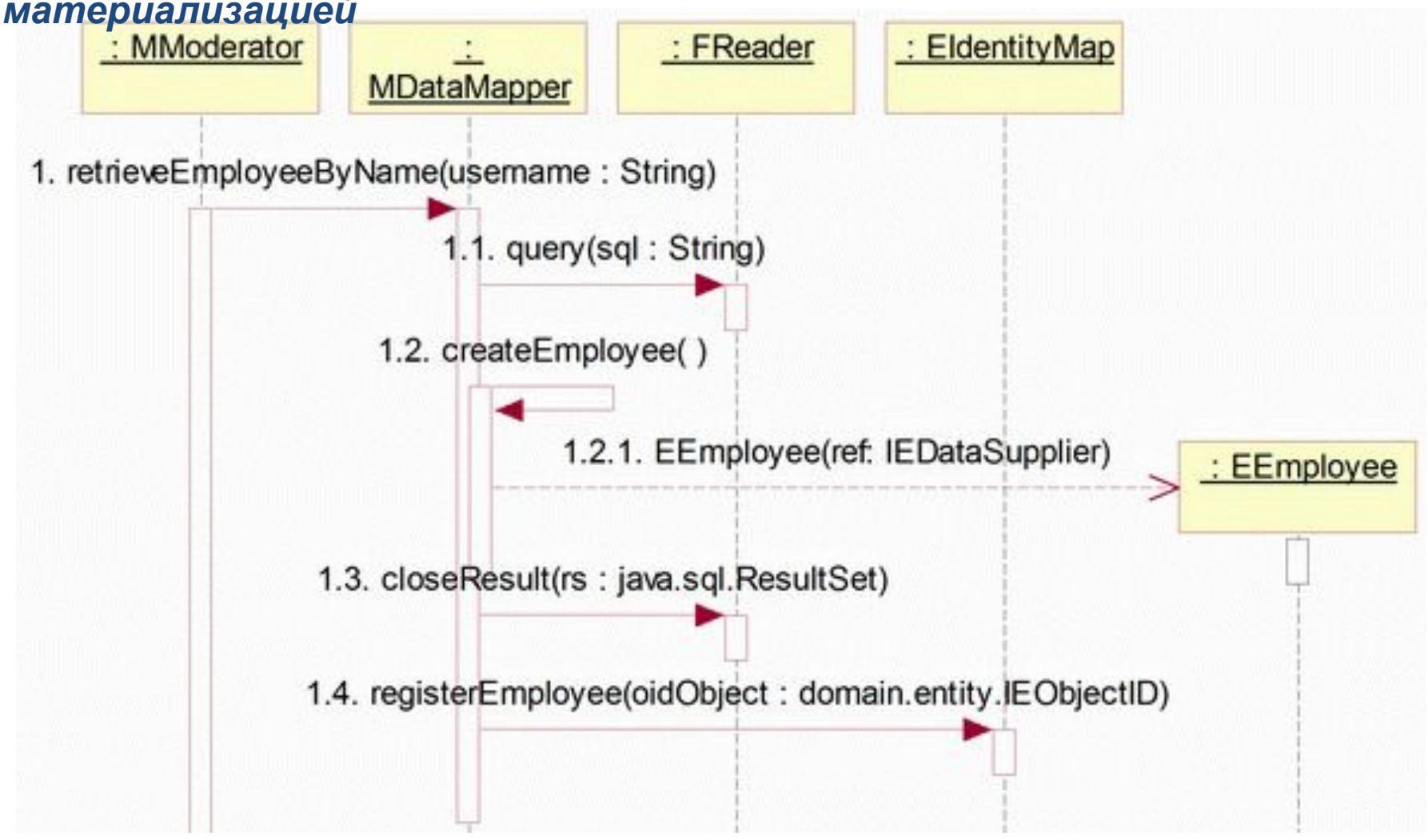
Рефакторинг класс извлечения позволяет извлечь обязанности ***Преобразователя дан-ных*** из класса MBroker в класс MDataMapper.

MDataMapper отде-ляет пакет entity от

Загрузка — импорт

Когда записи данных будут извлечены из БД, MDataMapper превращает их в объекты пакета entity. Затем новые объекты регистрируются в EIdentityMap.

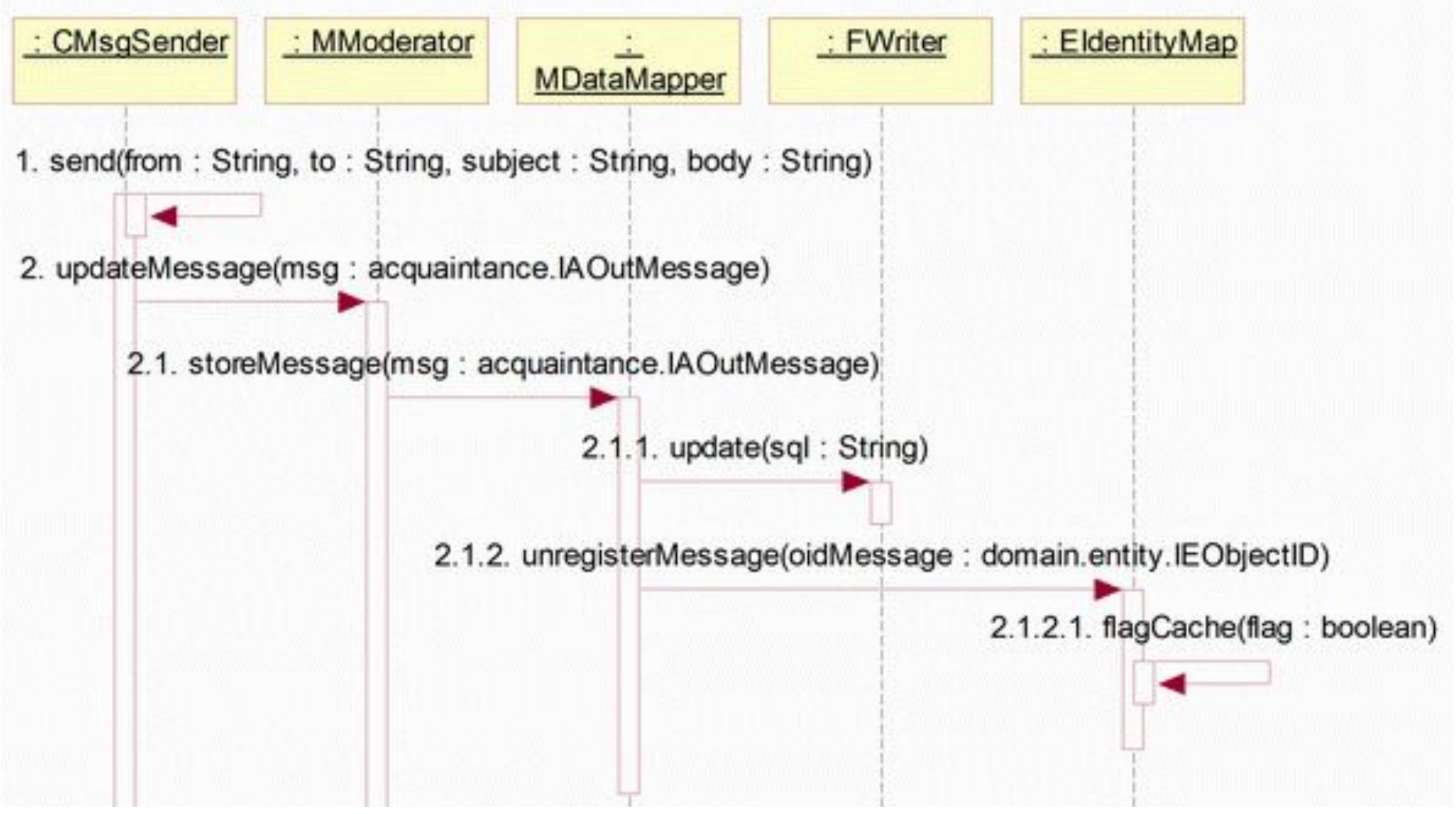
Это — операция загрузки. Процесс загрузки также называется ***материализацией***



Выгрузка — экспорт

Преобразователь данных должен обеспечивать и выгрузку объектов в БД.
Процесс выгрузки также известен как ***пассивация*** или ***дематериализация***.

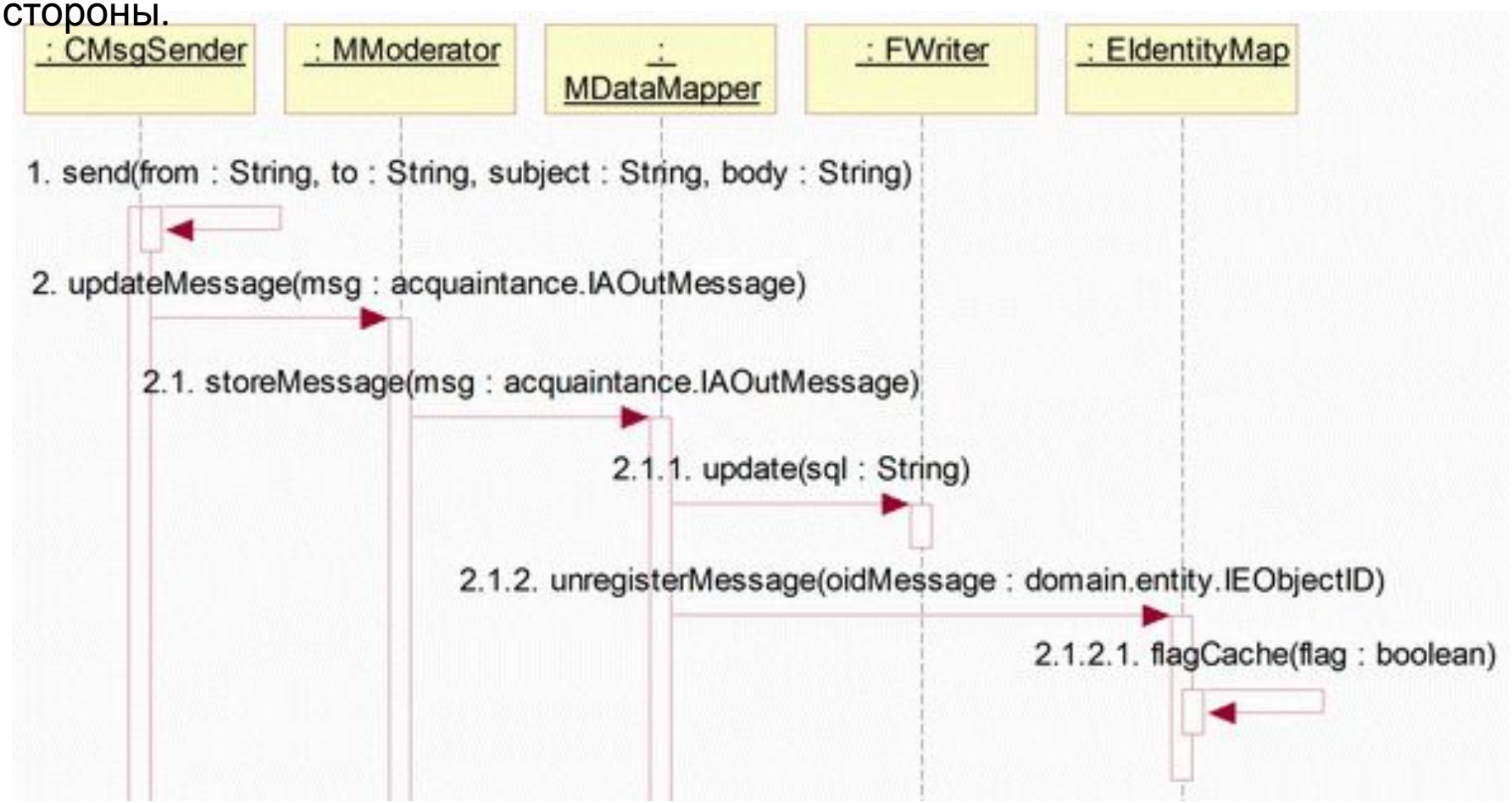
Выгрузка требуется, когда объект изменен, удален или создан новый .



Преобразователь данных MDataMapper

Используется единственный класс *Преобразователя данных* (MDataMapper), чтобы размещать и удалять строки данных у всех объектов пакета entity, известных в приложении (EEmployee, EContact и EOutMessage).

MModerator служит Фасадом для пакета mediator. MDataMapper обеспечивает ассоциации к EIdentityMap, с одной стороны, и к FReader и FWriter, с другой стороны.



Альтернативные стратегии Преобразователя данных

Более сложные системы или последовательные итерации могут требовать альтернативных решений, основанных на различных паттернах рефакторинга:

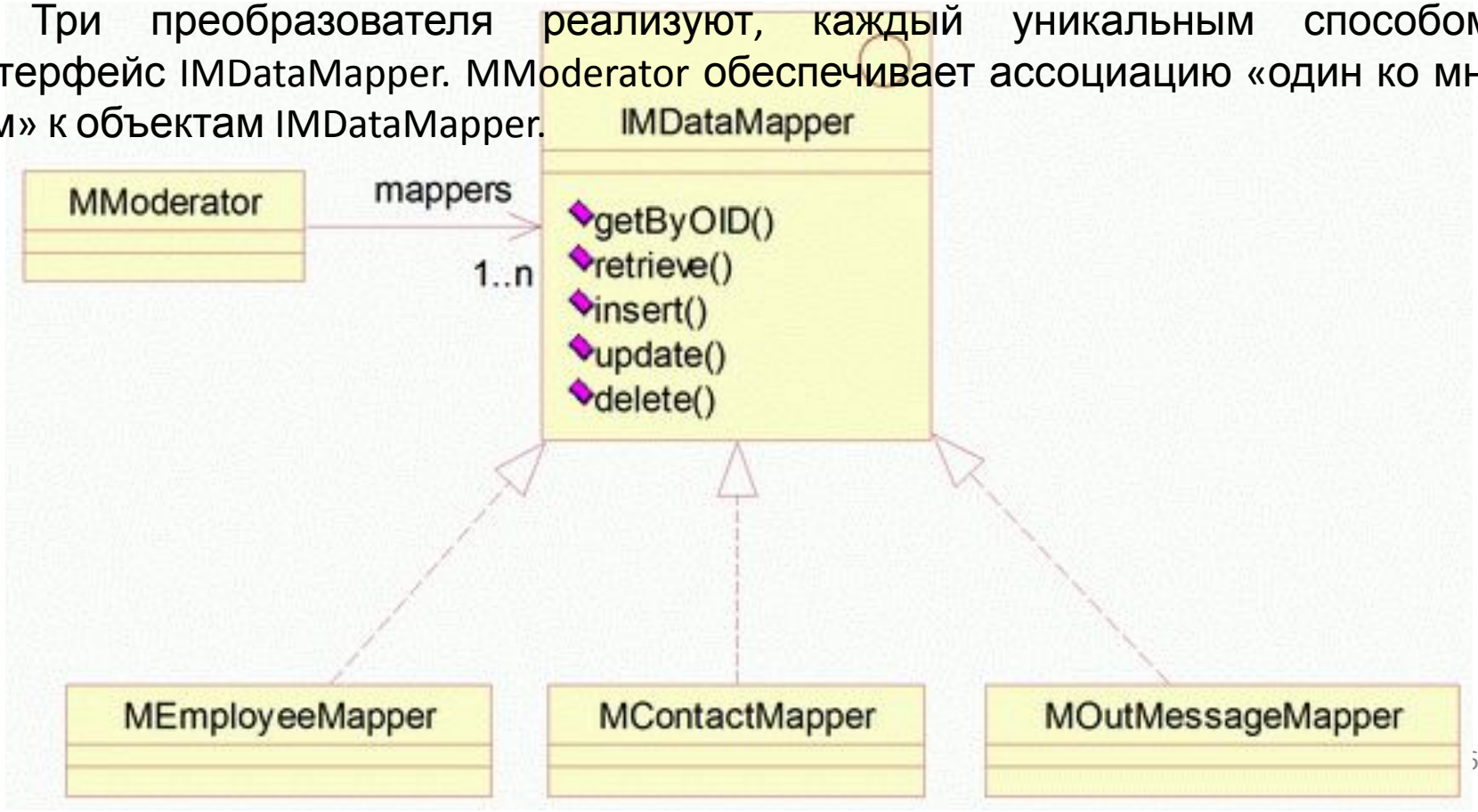
- ***Несколько Преобразователей данных***, возможно, по одному на каждый класс пакета entity.
- ***Использование метаданных*** и Java-отображения, чтобы динамически формировать в приложении *Преобразователи данных* по мере необходимости.
- ***Загрузка по требованию***, который размещает только данные, необходимые в настоящее время приложению, и, если возможно, создает пустые объекты для данных, не извлеченных из БД, но которые связаны с уже извлеченными данными.

Несколько Преобразователей данных

MDataMapper имел отдельные методы для управления запросами клиента, направленные на различные объекты пакета entity. В случае большого числа классов пакета entity, *Преобразователь данных* может стать слишком разрастись.

Альтернативное решение - один *Преобразователь данных* на каждый класс.

Три преобразователя реализуют, каждый уникальным способом, интерфейс IMDataMapper. MModerator обеспечивает ассоциацию «один ко мно гим» к объектам IMDataMapper.

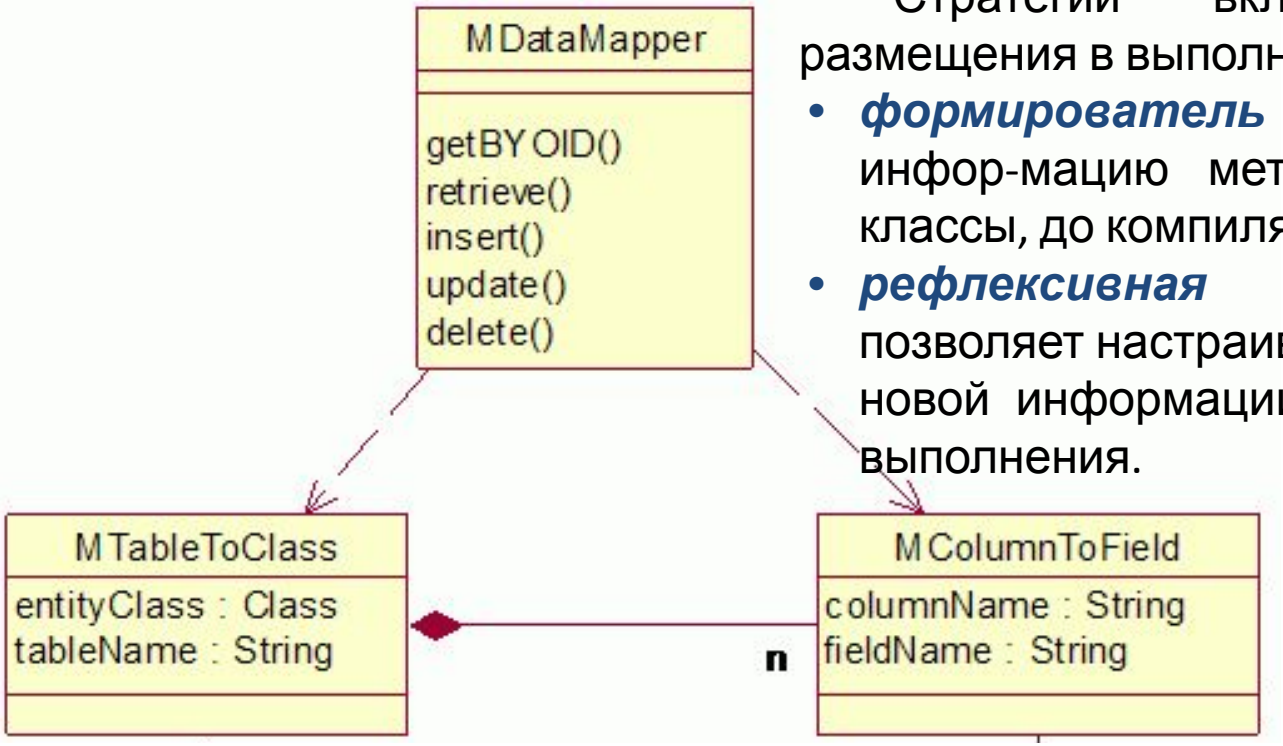


Преобразование метаданных

Цель паттерна ***Преобразователь метаданных*** — динамически формировать объектно-реляционное преобразование, основанное на метаданных.

Хранить метаданные можно:

- непосредственно ***в исходном коде*** приложения (в пакете mediator);
- ***во внешнем файле***, предпочтительно в XML-файле (приложение читает файл во время инициализации и создает соответствующие преобразования в структуре программы);
- ***в БД*** (одни и те же метаданные используются несколькими приложениями).



Стратегии включения информации размещения в выполняемом коде:

- ***формирователь кода***, который читает инфор-мацию метаданных и формирует классы, до компиляции программы;
- ***рефлексивная программа***, которая позволяет настраивать приложение к любой новой информации метаданных во время выполнения.

Загрузка по требованию (Lazy Load)

Основные виды операций извлечения:

Идентифицирующая загрузка (Identity load) — извлечение конкретных объектов, имеющих заданную величину OID или первичного ключа.

Загрузка на основе логического условия (Predicate load) — извлечение нескольких объектов, выполняя поиск по запросу на основе логических условий.

Основные стратегии загрузки:

Замкнутая загрузка (Closure load) — загружаются все объекты, доступные из указанного объекта.

Простая загрузка (Flat load) — загружаются только указанные объекты без объектов компонентов.

n-уровневая загрузка (n-levels load) — загружаются объекты, достижимые из указанного объекта в пределах данного числа уровней.

Замкнутая (активная) загрузка обычно неприемлема. Остальные стратегии - варианты *загрузки по требованию.*

Паттерн ***Загрузка по требованию*** определяется как «объект, который не содержит все данные, в которых вы нуждаетесь, но знает, как получить их».

Подходы, реализующие Загрузку по требованию:

- ***Инициализация по требованию (Lazy Initialization)***;
- ***Виртуальный заместитель (Virtual Proxy)***;
- ***Заместитель идентификатора объекта (OID Proxy)***.

Инициализация по требованию (Lazy Initialization)

По запросу от объекта-клиента *Преобразователь данных* ищет кэш данных и, если данных там нет, загружает их из БД.

Попытка *Преобразователя данных* получить данные из кэша может быть выполнена по значению элемента данных в объекте. Если значение null, то *Преобразователь данных* должен инициализировать элемент данных, обращаясь к БД.

Метод getContact() в EOutMessage

```
public IAContact getContact()
    { if (contactOID == null)
    contact = MDataMapper.retrieveContact(contactID);
    return contact;
}
```

Инициализация по требованию может использоваться в EOutMessage, если приложение содержит преобразователи данных, в которых имеется contactOID к загруженному объекту EContact, и элемент данных, содержащий contactID, который соответствует внешнему ключу в БД. Когда запрашивается getContact(), тот проверяет, является ли contactOID пустым. Если да, то посылается сообщение retrieveContact() объекту MDataMapper, чтобы за грузить его.

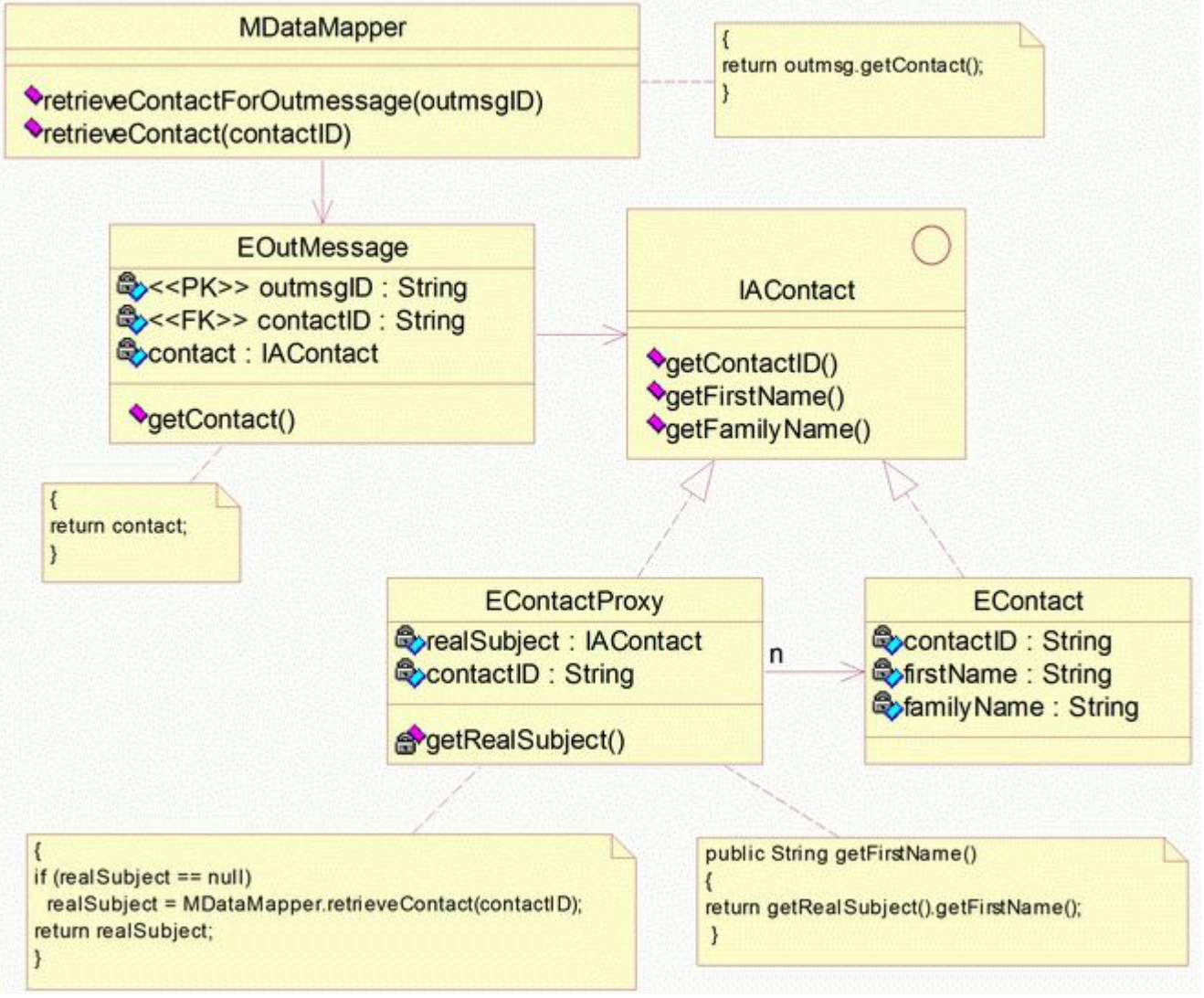
Виртуальный заместитель (Virtual Proxy)

В *Загрузке по требованию* заместитель обозначает объект, который может потребоваться загрузить при попытке обратиться к нему.

Заместитель заменяет объект и выступает в роли реального объекта. Он получает сообщения, предназначенные для реального объекта (который называется реальным субъектом в *Виртуальном заместителе)* и создает реальный объект, если он еще не существует.

Виртуальный заместитель (Virtual Proxy)

Предположим, что объекты пакета entity идентифицированы своими первичными ключами (PK).

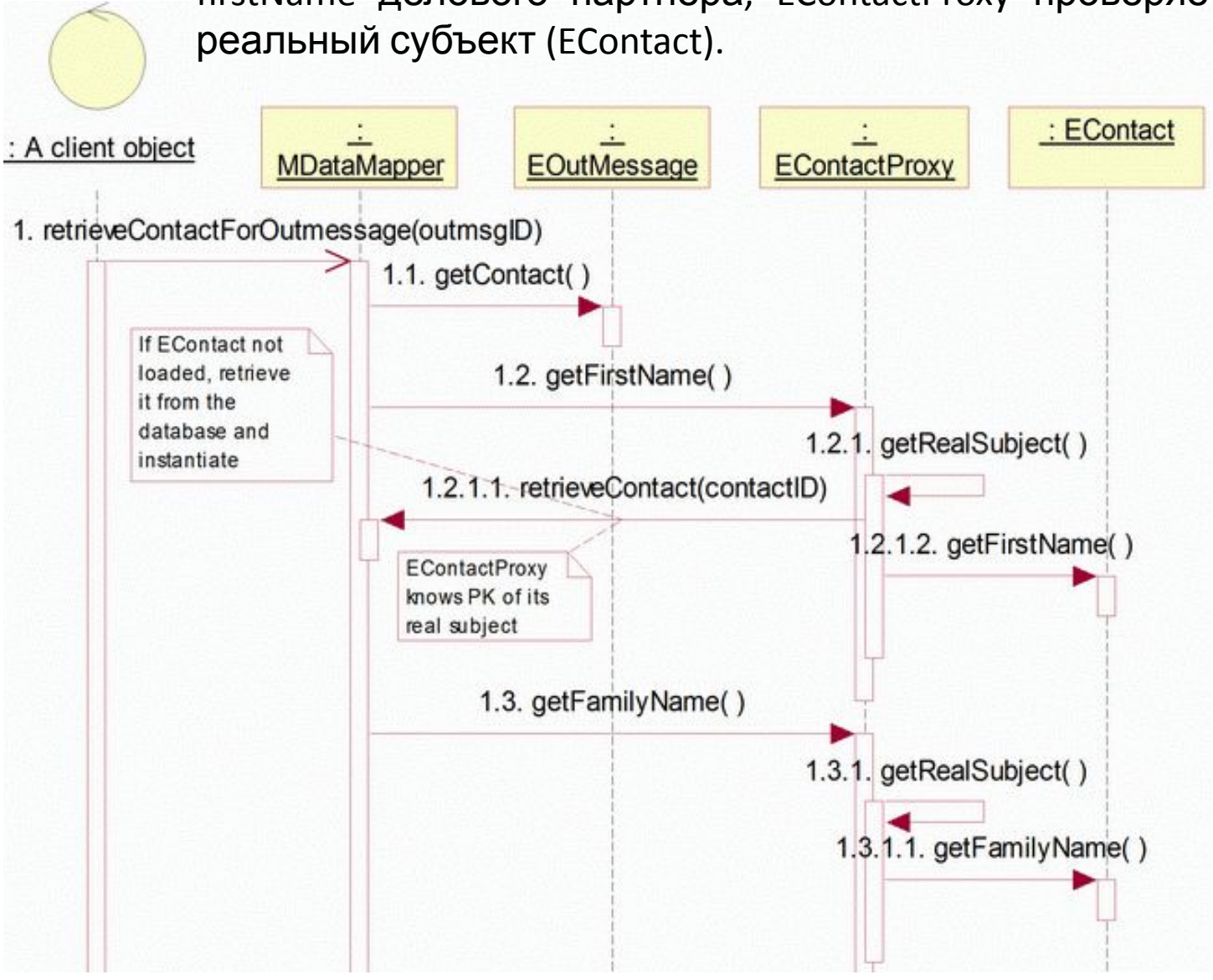


Предположим, что объект EOutMessage дол-жен быть загружен и что EContactProxy (класс «за-меститель делового парт-нера») знает PK своего реального субъекта. Когда MDataMapper соз-дает EContactProxy, он инициализирует его поле contactID соответствую-щим значением FK в EOutMessage.

Когда MDataMapper запрашивает делового партнера (getContact()), EOutMessage возвращает EContactProxy.

Виртуальный заместитель (Virtual Proxy)

Когда на следующем шаге MDataMapper запрашивает, например, firstName делового партнера, EContactProxy проверяет, существует ли реальный субъект (EContact).



Если его нет, запрашивается MDataMapper, чтобы инициализировать его. Затем EContactProxy может возвратить firstName.

Следующий запрос относитель-но familyName, выполняется подоб-ным же образом за исключением того, что EContact уже загружен.

Заместитель идентификатора объекта (OID Proxy)

Одноэлементный класс (EIdentityMap), который обеспечивает задание OID объектам выполняет функции замещающих классов. Такой подход может быть зарегистрирован в паттерне — *паттерне Заместитель идентификатора объекта*.

EIdentityMap знает, загружен ли объект пакета entity. После загрузки объекту задается OID, и все его элементы данных инициализируются. Инициализация включает значения внешних ключей (FK), полученные из БД. Основанные на OID связи ассоциации, соответствующие внешним ключам, инициализируются значением null, если связанный объект не был еще загружен.

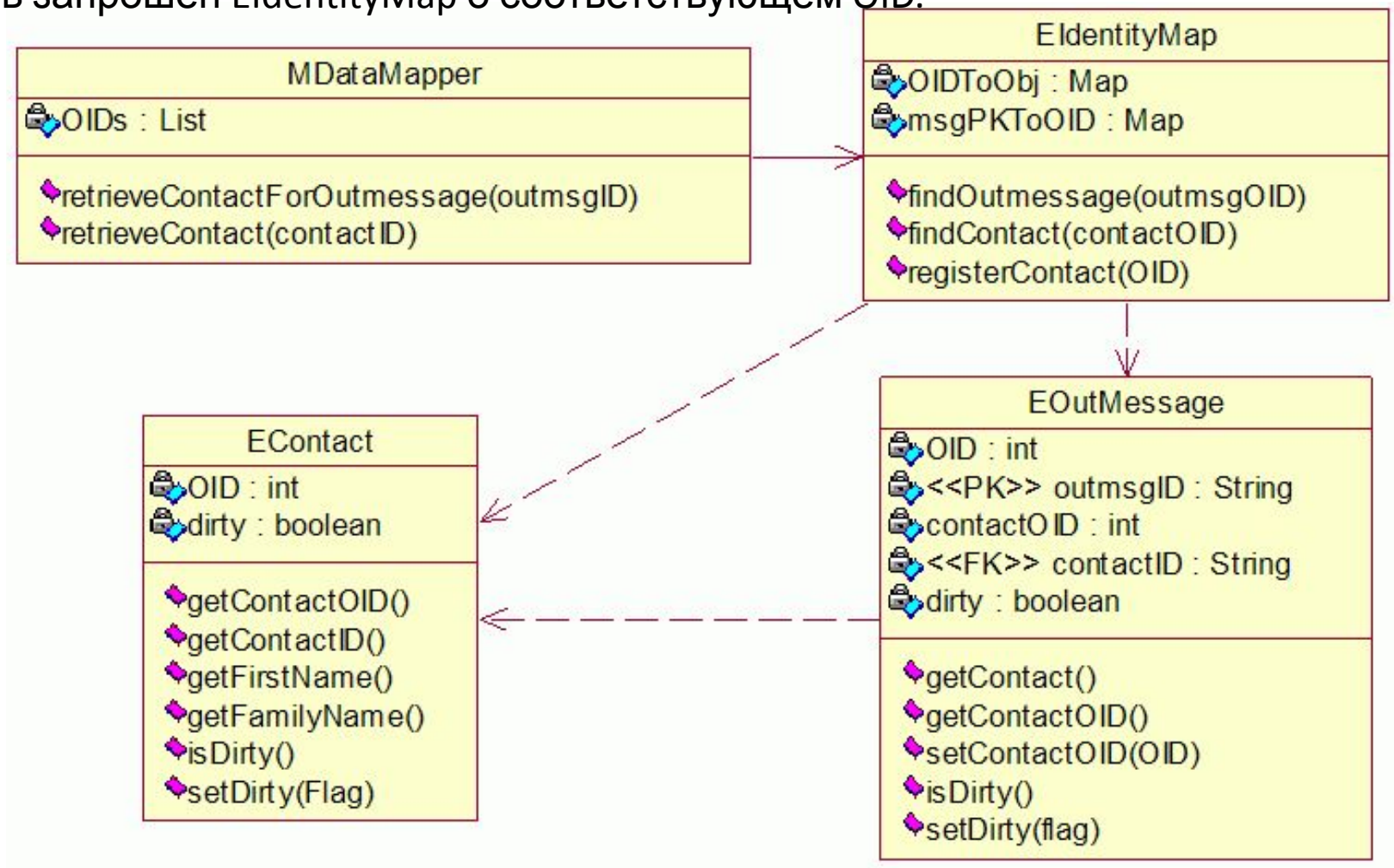
Дополнительное преимущество *Заместителя идентификатора объекта* над *Виртуальным заместителем* — легкость, с которой может быть опреде лено измененное/не измененное состояние объекта пакета entity. Для объекта пакета entity, загруженного первым, устанавливается флаг неизмененного объекта. Если содержание данных объекта выходит из синхронизма со своим соответствующим содержанием в БД, flag (флаг) устанавливается в состояние «измененный». Объект пакета entity всегда знает свое состояние и может вызывать перезагрузку из БД, чтобы обновить содержание.

Имеются две разновидности *Заместителя идентификатора объекта*:

- ***навигация по коллекции идентичности объектов***;
- ***навигация по классам пакета entity***.

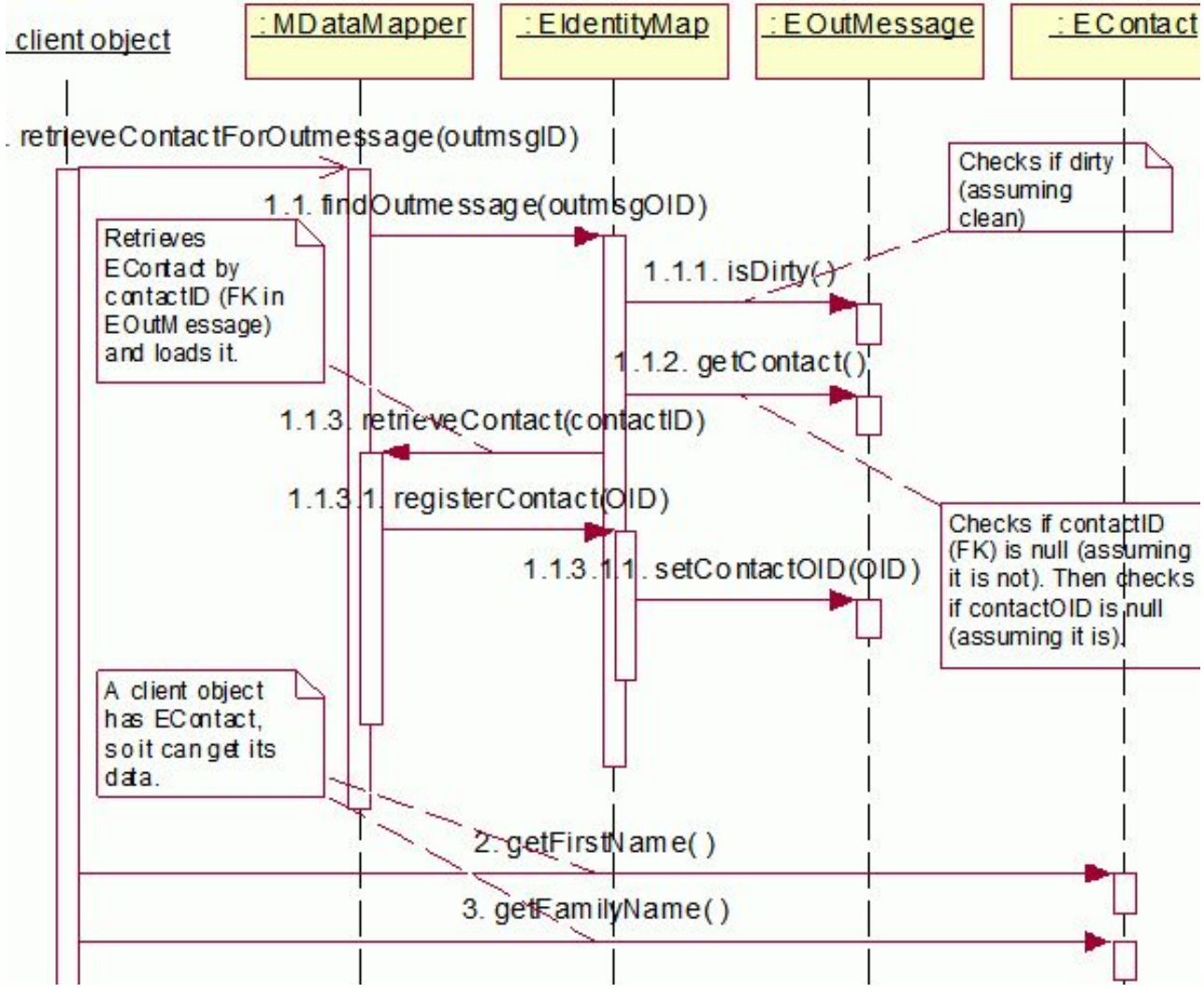
Навигация по коллекции идентичности объектов

Паттерн Заместитель идентификатора объекта. contactOID класса EOutMessage служит заместителем объекта Econtact. MDataMapper содержит список OID всех объектов, загруженных в память. По значению РК объекта (например, outmsgID), может быть запрошен EIdentityMap о соответствующем OID.



Навигация по коллекции идентичности

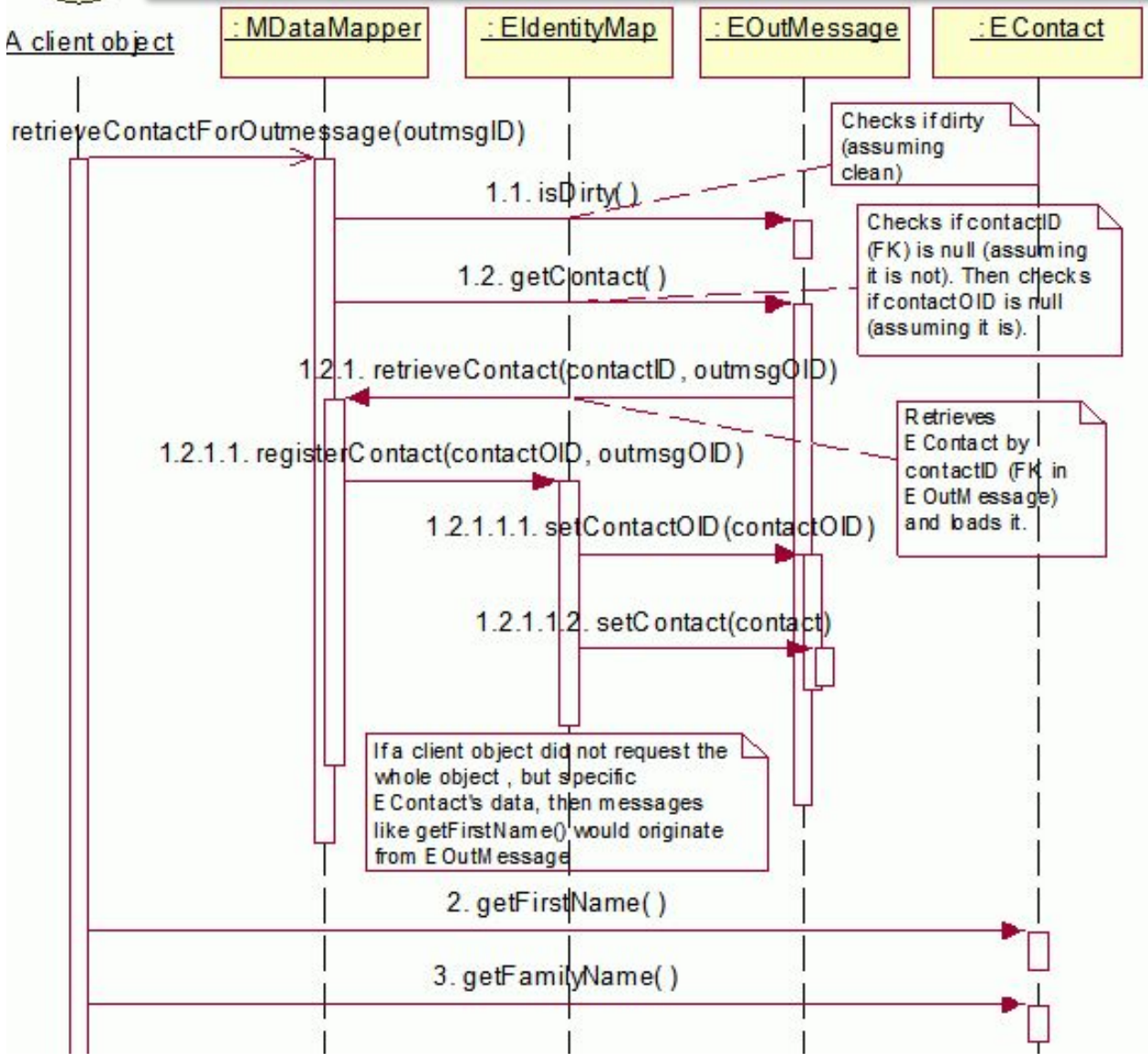
Объекты пакета entity содержатся в коллекциях EIdentityMap. Они имеют только OID к другим объектам.



Каждый объект пакета entity знает изменен он или нет. Это проверяется методом isDirty(). EIdentityMap запрашивает getContact() у EOutMessage. EOutMessage проверяет, является ли значение внешнего ключа к EContact пустым и равен ли contactOID null. Если да, то EContact не был загружен. EIdentityMap запрашивает MDataMapper, извлечь EContact по значению FK. Econtact загружается. MDataMapper корректирует свой список OID. EIdentityMap корректирует свои коллекции, а в EOutMessage устанавлива-ется contactOID. EContact возвращается к client object, который может теперь непосредственно запросить его данные (getFirstName(), getFamilyName() и т.д.)

Навигация по классам

Навигация по классам требует, чтобы объект пакета entity не содержал лишь OID к другому объекту, а содержал сам этот объект.



После определения, что EOutMessage не изменен, MDataMapper сообщает это методу getContact(). Если EContact не загружен, EOutMessage инициирует загрузку. Он передает свой outmsgOID объекту MDataMapper, чтобы новый EContact мог быть связан с ним, когда будет зарегистрирован в EIdentityMap.

Логика по загрузке EContact заметно переместилась к EOutMessage.

Единица работы (Unit of Work)

Объекты пакета entity загружаются если они: 1) не были загружены, 2) стали измененными. Для упрощения итерация 1 объявляет весь кэш измененным, как только будет передано по электронной почте любое EOutMessage. Флаг измененного состояния находится в классе MBroker.

После рефакторинга флаг измененного состояния появляется в каждом объекте.

При загрузке флаг объекта очищается. Флаг устанавливается измененным, когда объект изменится.

Метод updateMessage() в MModerator

```
87: public boolean updateMessage(IAOutMessage msg) {
91: msg.setSentDate(new Java.sql.Date(System.currentTimeMillis()));
92:
93:     boolean b = mapper.storeMessage(msg);
94: if (b) {
96:     msg.setContact (null);
97:     msg.setCreatorEmployee(null);
98:     msg.setSenderEmployee(null);
99:     msg.setDirty(true);
100:    }
```

После того как сообщение EOutMessage будет передано по электронной почте, MModerator посылает сообщение setDirty() измененному объекту EOutMessage (стр.99).

Единица работы (Unit of Work)

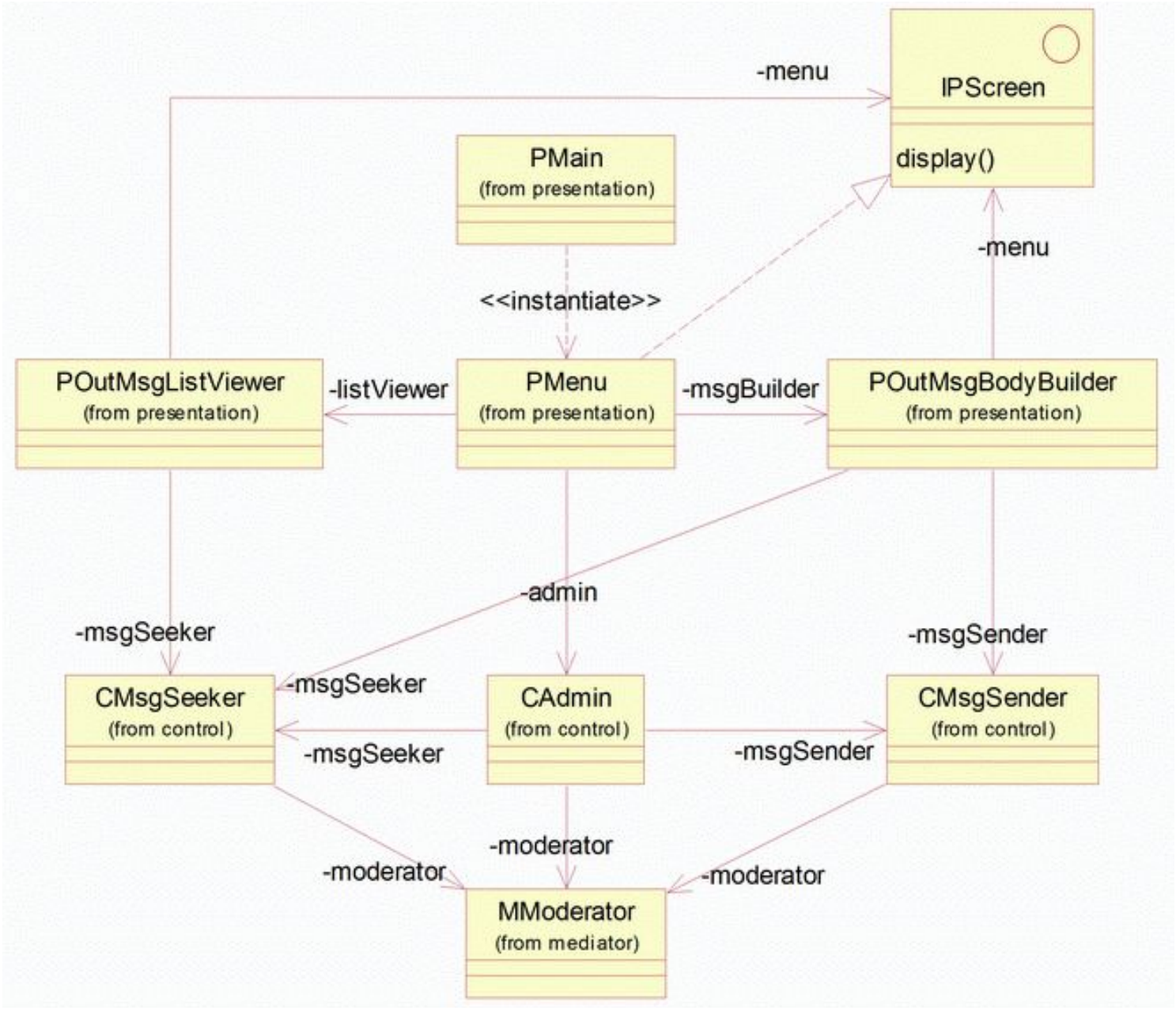
Подход, где каждый объект пакета entity знает, является ли он измененным или нет, позволяет осуществить индивидуальную загрузку и выгрузку объектов. Однако многие бизнес-транзакции, управляемые приложением, делают многочисленные изменения у разнообразных объектов в кэше перед попыткой записать эти изменения в БД. Измененные объекты должны быть со ответственно помечены как измененные, когда БД подтверждает, что транзакция успешно завершена. Эти проблемы быстро становятся очень сложными.

Паттерн ***Единица работы*** «обслуживает список объектов, на которые воздействует бизнес-транзакция, и координирует перезапись изменений и решение проблем параллелизма». Изменения могут быть следствием трех операций модификации: размещения нового объекта в БД, удаления объекта из БД или обновления элементов данных объекта.

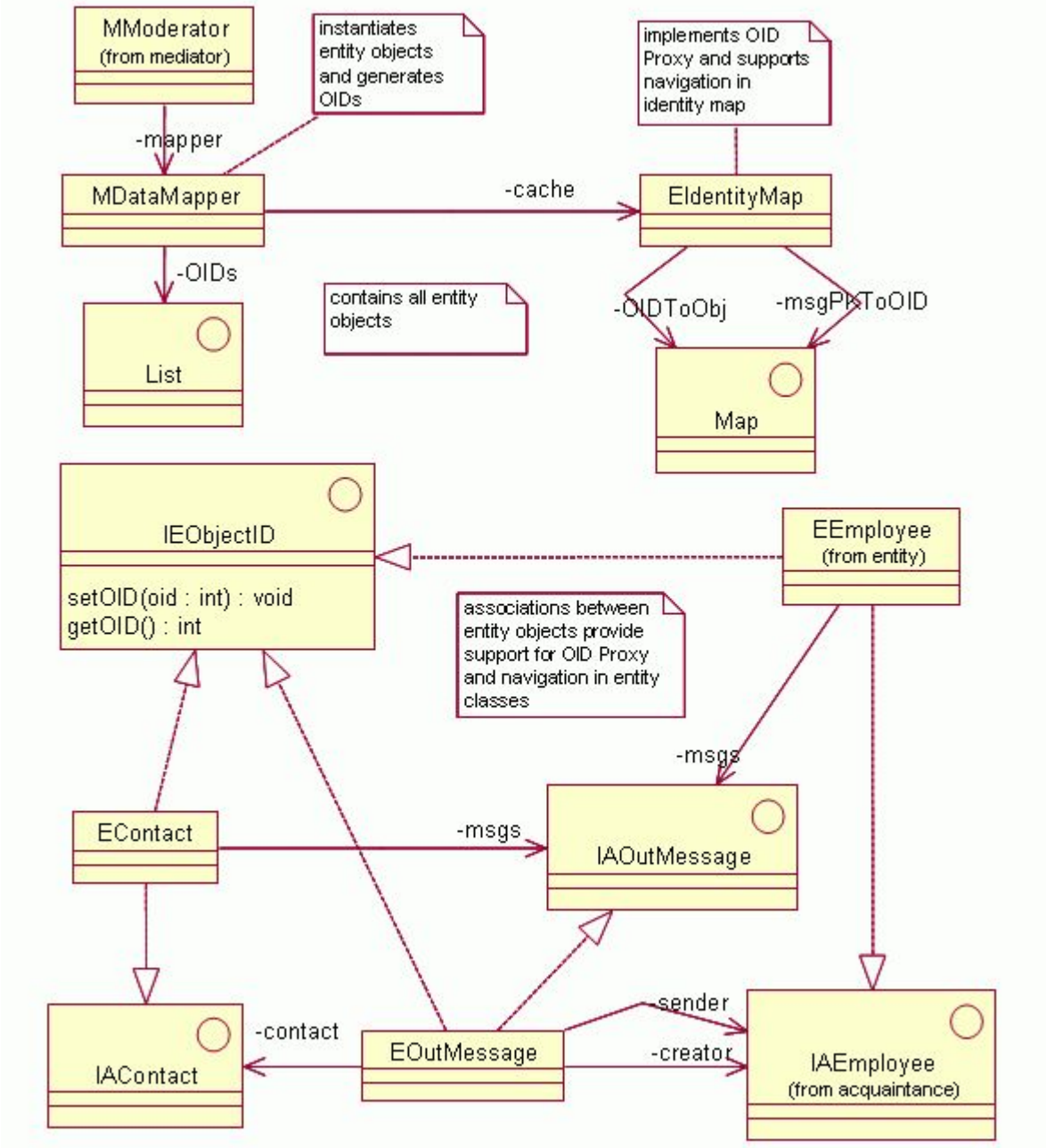
Паттерн *Единица работы* требует создания нового класса в пакете mediator. Класс можно назвать MWorkUnit.

Улучшенная модель классов

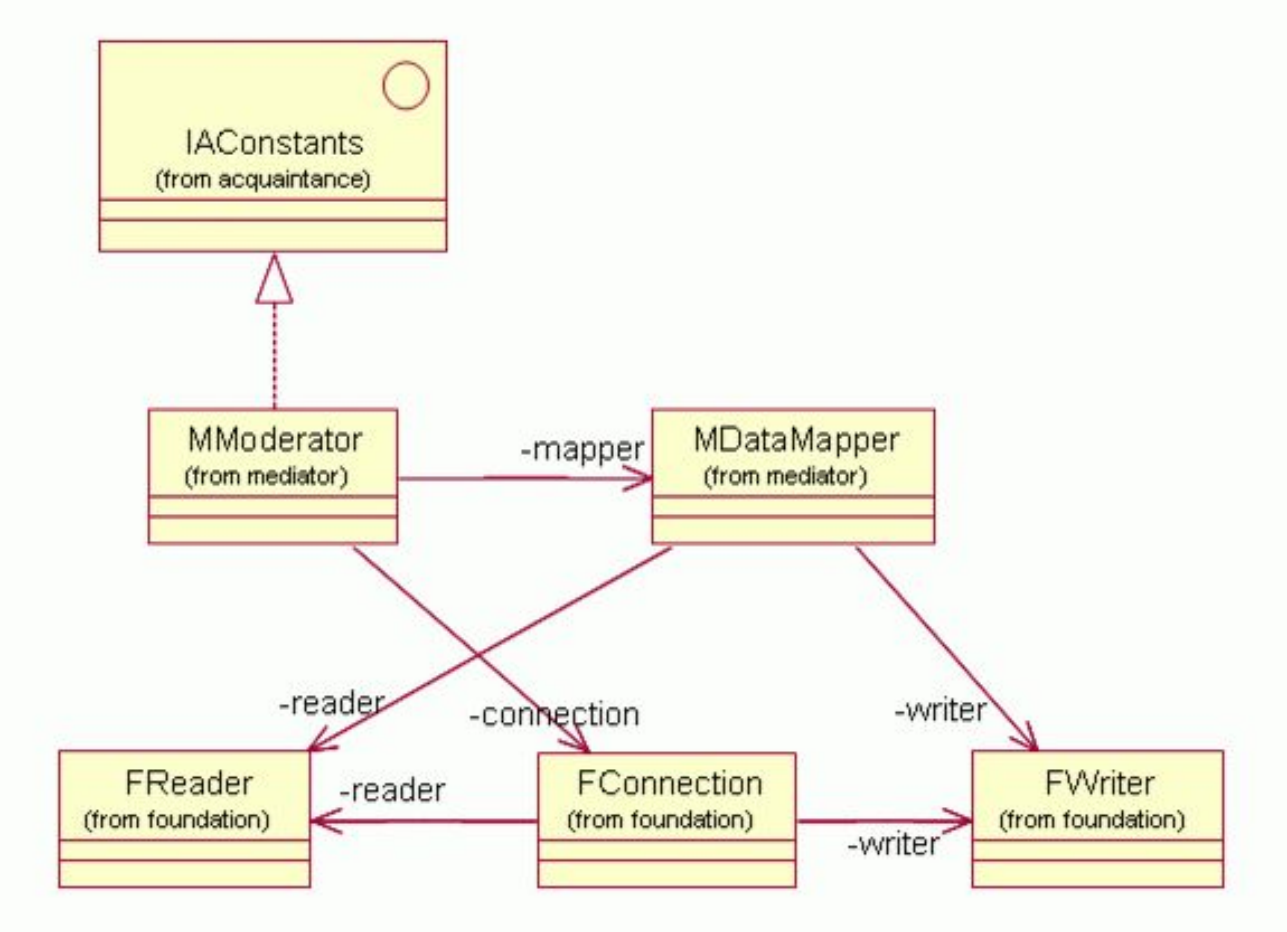
Слои presentation и control



Слой domain


MModerator
(from mediator)
instantiates
entity objects
and generates
OIDs
implements OID
Proxy and supports
navigation in
identity map
-mapper
MDataMapper
-cache
EIdentityMap
-OIDs
contains all entity
objects
-OIDToObj
-msgPKToOID
List
Map
IEObjectID
setOID(oid : int) : void
getOID() : int
EEmployee
(from entity)
associations between
entity objects provide
support for OID Proxy
and navigation in entity
classes
-msgs
EContact
-msgs
IAOutMessage
-contact
IAContact
EOutMessage
sender
-creator
IAEmployee
(from acquaintance)

Пакеты mediator и foundation



Резюме

1. ***Рефакторинг*** — процесс чистки и улучшения внутренней структуры кода без изменения его внешнего поведения.
2. ***Методы рефакторинга*** — основные принципы и лучшая практика изменения кода для улучшения возможности его сопровождения (понятности, удобства сопровождения и масштабируемости).
3. Рефакторинг ***Класс извлечения*** расщепляет большой класс на ряд меньших классов.
4. Рефакторинг ***Метод извлечения*** преобразует дублированный код в отдельный метод.
5. Рефакторинг ***Метод подключения*** устраняет метод включением его функциональных возможностей в другой существующий метод.
6. Рефакторинг ***Интерфейс извлечения*** выделяет множество сигнатур методов, дублированных в нескольких классах или используемых несколькими клиентами, в интерфейс.
7. ***Паттерны рефакторинга*** — структурные паттерны, используемые в рефакторинге кода.
8. Паттерн ***Коллекция идентичности объектов*** назначает идентификаторы объектам и поддерживает коллекции, чтобы найти объекты, находящиеся в памяти, на основе их идентификаторов.
9. Паттерн ***Преобразователь данных*** отделяет объекты, находящиеся в памяти, от их представления в БД и отвечает за поддержание кэшей памяти объектов.

Резюме

10. ***Загрузка*** (импорт, материализация) — процесс извлечения записей из БД и преобразования их в объекты памяти. Противоположная операция называется *выгрузкой* (экспортом, пассивацией, дематериализацией).
11. Паттерн ***Загрузка по требованию*** загружает только отобранные объекты из БД в память, но он может загружать оставшиеся и связанные объекты, когда это необходимо. Подходами к Загрузке по требованию являются Инициализация по требованию, Виртуальный заместитель и Заместитель идентификатора объекта.
12. Паттерн ***Инициализация по требованию*** загружает объекты по определенному запросу от объекта-клиента, ответственного за поддержание кэша пакета entity. Загрузка для клиентов не прозрачна.
13. Паттерн ***Виртуальный заместитель*** использует объект-заместитель, который является заменой реального объекта и может загрузить реальный объект способом, прозрачным для клиента.
14. Паттерн ***Заместитель идентификатора объекта*** — удобная замена паттерна Виртуальный заместитель в приложениях, которые поддерживают коллекции OID объектов. Коллекции могут использоваться вместо классов-заместителей. Имеются две разновидности Заместителя идентификатора объекта, которые отличаются тем, как программа реализует навигацию между связанными объектами: Навигация по коллекции идентичности объектов и Навигация по классам пакета entity.

Резюме

16. ***Навигация по коллекции идентичности объектов*** использует класс Коллекция идентичности объектов каждый раз, когда приложение должно осуществить навигацию к связанному объекту пакета entity (то есть связанному ссылочной целостностью). Если объект X связан с объектом Y, Коллекция идентичности объектов должна запросить X, чтобы получить связь с Y, и затем осуществляет доступ к Y.
17. ***Навигация по классам пакета entity*** использует классы пакета entity, чтобы осуществить навигацию по связанным классам. Если объект X связан с объектом Y, Коллекция идентичности объектов позволила бы клиенту осуществить доступ к X, но затем передает управление к X, чтобы продолжить навигацию к Y.
18. Паттерн ***Единица работы*** обеспечивает приложение знанием проблем о бизнес-транзакциях и параллелизме. Он хранит последовательность изменений объектов пакета entity и то, действительно ли изменения были переданы в БД.

Литература

1. Практическая программная инженерия на основе учебного примера / Л.А. Мацяшек, Б.Л. Лионг. – М.: БИНОМ. Лаборатория знаний, 2009. – 956 с.

2. Фаулер М. Рефакторинг: улучшение существующего кода. – СПб.: Символ-Плюс, 2002.

3. Фаулер М. Архитектура корпоративных программных приложений. – М.: Издат. дом «Вильямс», 2006. – 544 с.

4. Ларман К. Применение UML и шаблонов проектирования. 2-е издание. – М.: Издат. дом «Вильямс», 2002. – 624 с.

5. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного программирования. Паттерны проектирования.- СПб.: Питер, 2006

6. Шаблоны [illegible] http://oodesign.com/Patterns