Дисциплина

СПЕЦИФИКАЦИЯ, ПРОЕКТИРОВАНИЕ И АРХИТЕКТУРА ПРОГРАММНЫХ СИСТЕМ

Тема 08

АРХИТЕКТУРА ИНФОРМАЦИОННЫХ СИСТЕМ. ТИПОВЫЕ РЕШЕНИЯ.

Преподаватель

к.т.н. Романенко Сергей Александрович

Цель проектирования

Создание системы, которая:

- удовлетворяет заданным (возможно, неформальным) функциональным спецификациям;
- согласована с ограничениями, накладываемыми оборудованием;
- удовлетворяет явным и неявным требованиям по эксплуатационным качествам и потреблению ресурсов;
- удовлетворяет явным и неявным критериям дизайна продукта;
- удовлетворяет требованиям к самому процессу разработки, таким, например, как продолжительность и стоимость, а также привлечение дополнительных инструментальных средств.

Архитектура системы

Проектирование — выявление ясной и относительно простой внутренней структуры, называемой ***архитектурой системы***.

Архитектура - *это базовая* ***организация системы****, воплощенная в ее* ***компонентах****, их* ***отношениях*** *между собой и с* ***окружением****, а также* ***принципы****, определяющие* ***проектирование и развитие*** *системы*

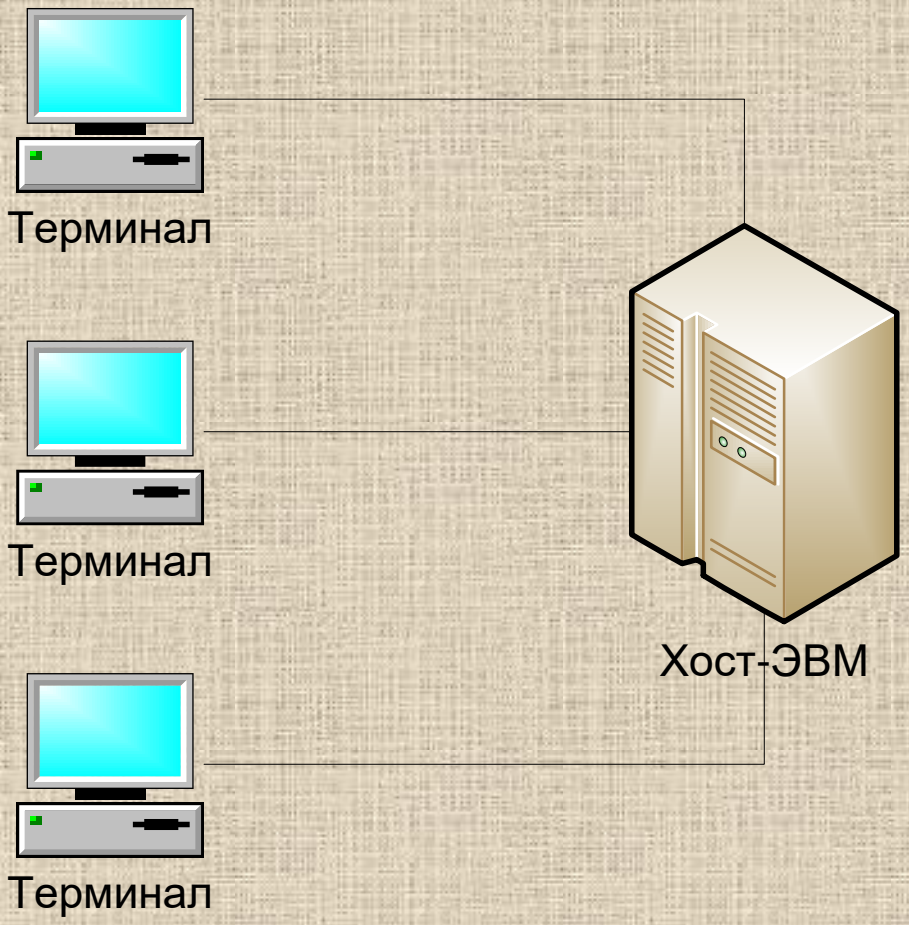
Характеристики корпоративных систем

1. Предназначены для автоматизации деятельности организации или предприятия, а не отдельных задач или процессов.
2. Подразумевают необходимость долговременного хранения данных.
3. Большой объем хранимых и обрабатываемых данных.
4. Многопользовательский режим работы.
5. Многовариантный оконный интерфейс пользователя.
6. Интеграция с другими системами.
7. Реализация бизнес-логики, иногда слабо формализованной

Классификация информационных систем по архитектуре

1. Централизованная архитектура
2. Архитектура "файл-сервер"
3. Двухзвенная архитектура "клиент-сервер"
4. Многозвенная архитектура "клиент-сервер"
5. Архитектура распределенных систем
6. Архитектура Веб-приложений
7. Сервис-ориентированная архитектура
8. Архитектура облачных вычислений

Централизованная архитектура



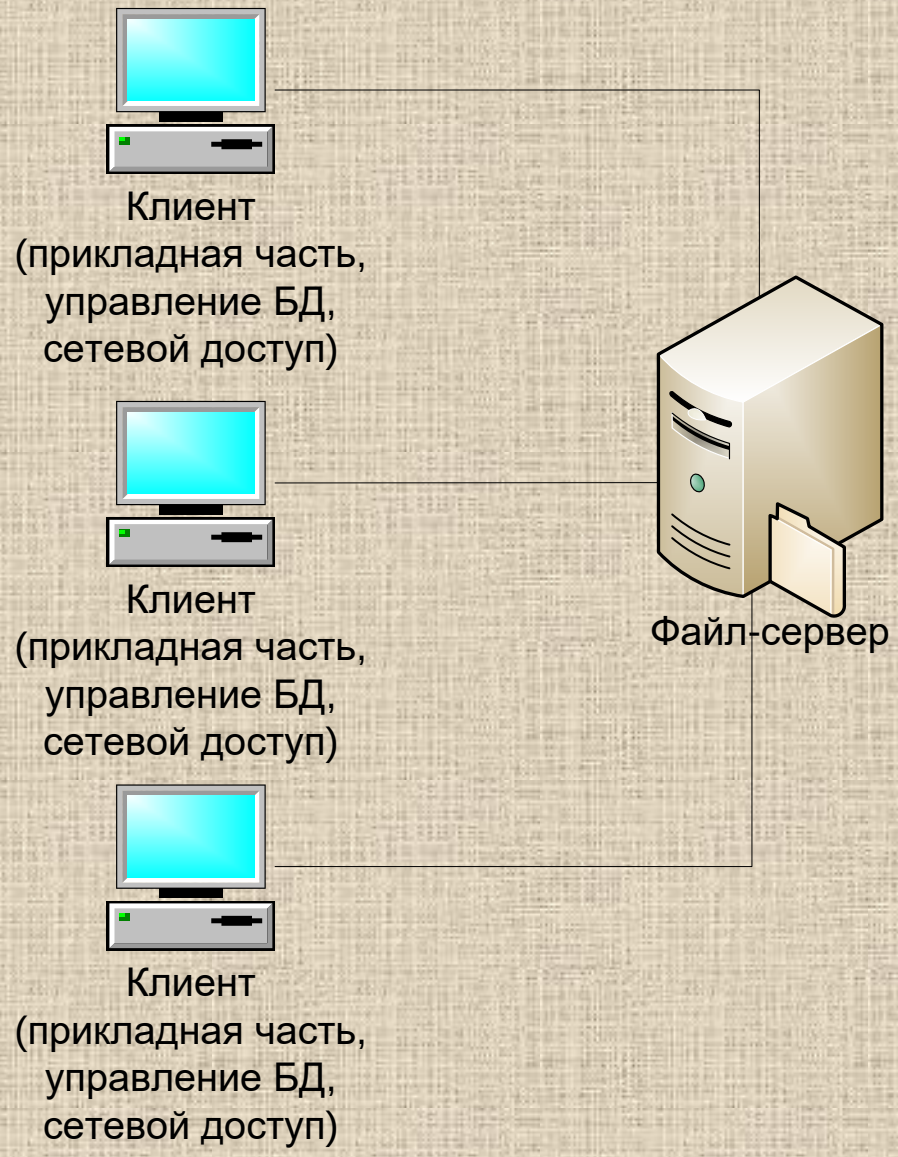
Распространена в 70-х и 80-х годах прошлого века и реализовывалась на базе мейнфреймов (например, IBM-360/370 или их отечественных аналогов серии ЕС ЭВМ). Характерная особенность такой архитектуры – полная "неинтеллектуальность" терминалов. Их работой управляет хост-ЭВМ

Централизованная архитектура

- ✓ **Достоинства:**
  - пользователи совместно используют дорогие ресурсы ЭВМ и дорогие периферийные устройства
  - централизация ресурсов и оборудования облегчает обслуживание и эксплуатацию вычислительной системы
  - отсутствует необходимость администрирования рабочих мест пользователей

- ✓ **Главный недостаток:**
  - пользователи полностью зависят от администратора хост-ЭВМ

Архитектура «файл-сервер»



Файл-серверные приложения – приложения, схожие по своей структуре с локальными приложениями и использующие сетевой ресурс для хранения программы и данных.

Функции сервера: хранения данных и кода программы.

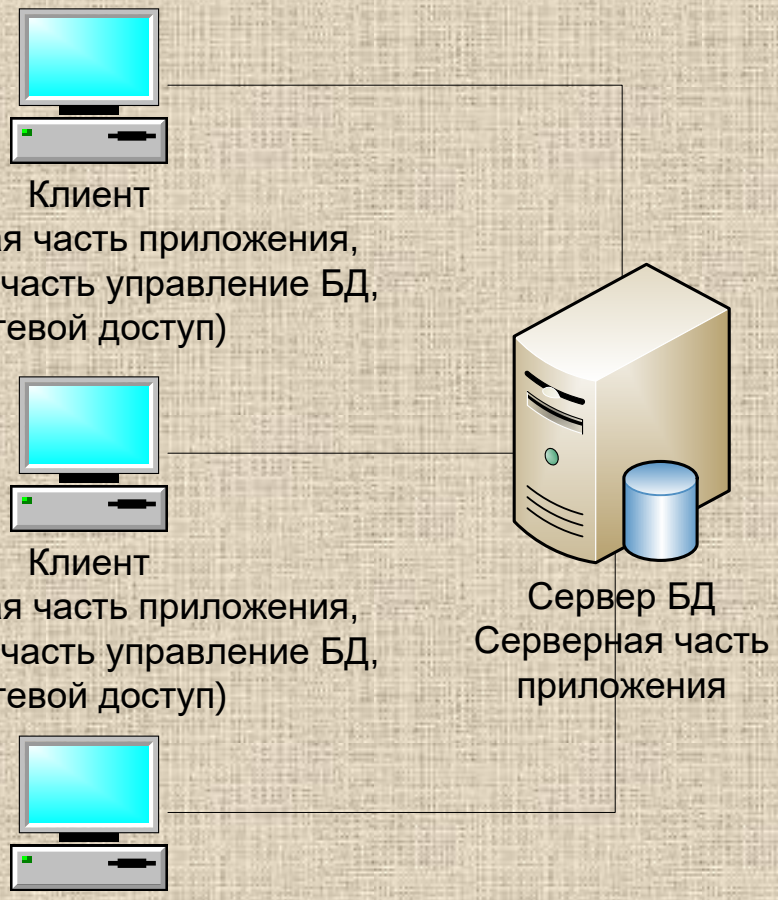
Функции клиента: обработка данных происходит исключительно на стороне клиента.

Архитектура «файл-сервер»

- **Достоинства:**
  - многопользовательский режим работы с данными
  - удобство централизованного управления доступом
  - низкая стоимость разработки
  - высокая скорость разработки
  - невысокая стоимость обновления и изменения ПО

- **Недостатки:**
  - проблемы многопользовательской работы с данными
  - низкая производительность
  - плохая возможность подключения новых клиентов
  - ненадежность системы

Двухуровневая архитектура «клиент-сервер»

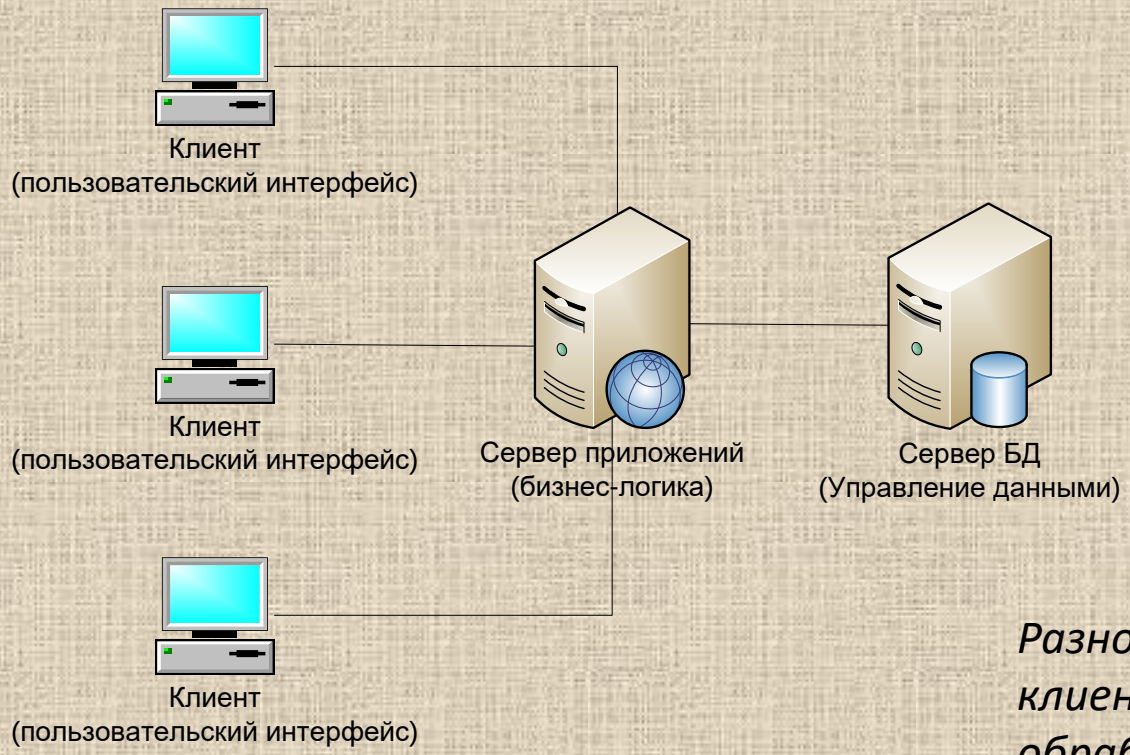


Клиент-сервер (Client-server) – вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг (сервисов), называемых серверами, и заказчиками услуг, называемых клиентами. Нередко клиенты и серверы взаимодействуют через компьютерную сеть и могут быть как различными физическими устройствами, так и программным обеспечением

Двухуровневая архитектура «клиент-сервер»

- **Достоинства:**
  - возможность распределить функции вычислительной системы между несколькими независимыми компьютерами
  - все данные хранятся на защищенном сервере
  - поддержка многопользовательской работы
  - гарантия целостности данных
- **Недостатки:**
  - неработоспособность сервера может сделать неработоспособной всю вычислительную сеть
  - сложное администрирование
  - высокая стоимость оборудования
  - бизнес логика приложений осталась в клиентском ПО

Многоуровневая архитектура «клиент-сервер»

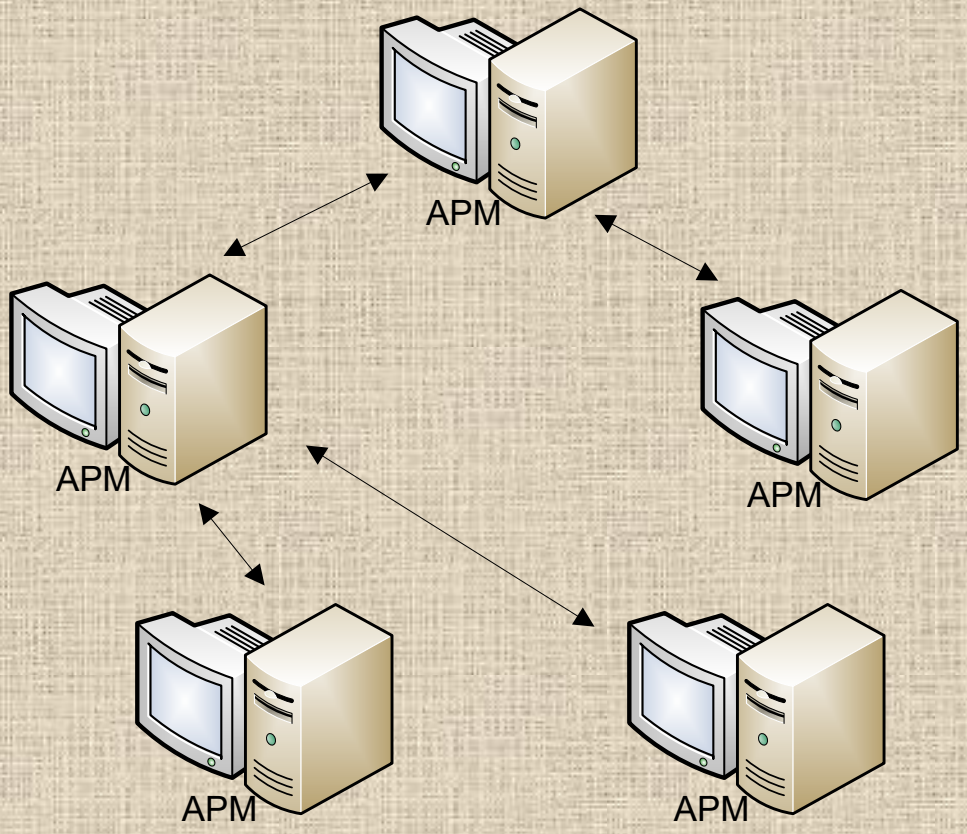


Разновидность архитектуры клиент-сервер, в которой функция обработки данных вынесена на один или несколько отдельных серверов. Это позволяет разделить функции хранения, обработки и представления данных для более эффективного использования возможностей серверов и клиентов.

Многоуровневая архитектура «клиент-сервер»

- **Достоинства:**
  - клиентское ПО не нуждается в администрировании
  - масштабируемость
  - конфигурируемость
  - высокая безопасность и надежность
  - низкие требования к скорости канала между терминалами и сервером приложений
  - низкие требования к производительности и техническим характеристикам терминалов
- **Недостатки:**
  - сложность администрирования и обслуживания
  - более высокая сложность создания приложений
  - высокие требования к производительности серверов приложений и сервера базы данных
  - высокие требования к скорости канала (сети) между сервером базы данных и серверами приложений

Архитектура распределенных систем

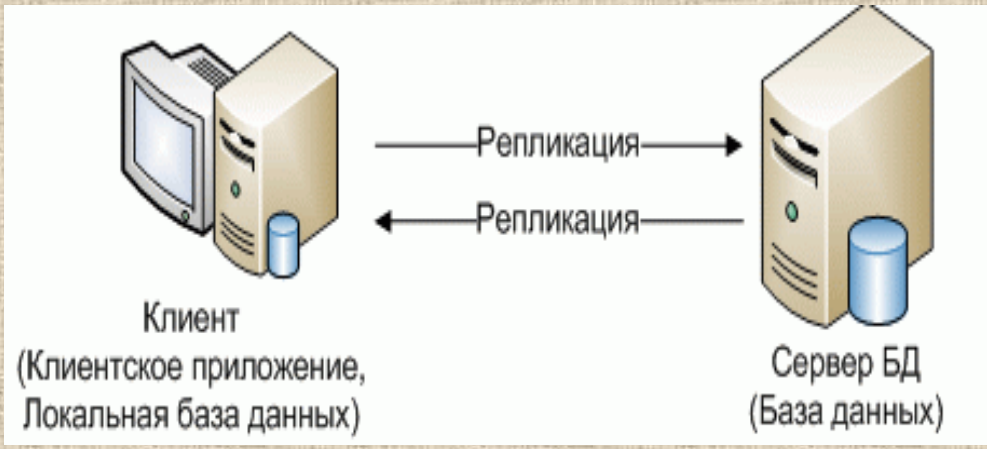


Суть распределенной системы заключается в том, чтобы хранить локальные копии важных данных.

Каждый АРМ независим, содержит только ту информацию, с которой должен работать, а актуальность данных во всей системе обеспечивается благодаря непрерывному обмену сообщениями с другими АРМами.
Обмен сообщениями между АРМами может быть реализован различными способами, от отправки данных по электронной почте до передачи данных по сетям.

Такая архитектура системы также позволяет организовать распределенные вычисления между клиентскими машинами

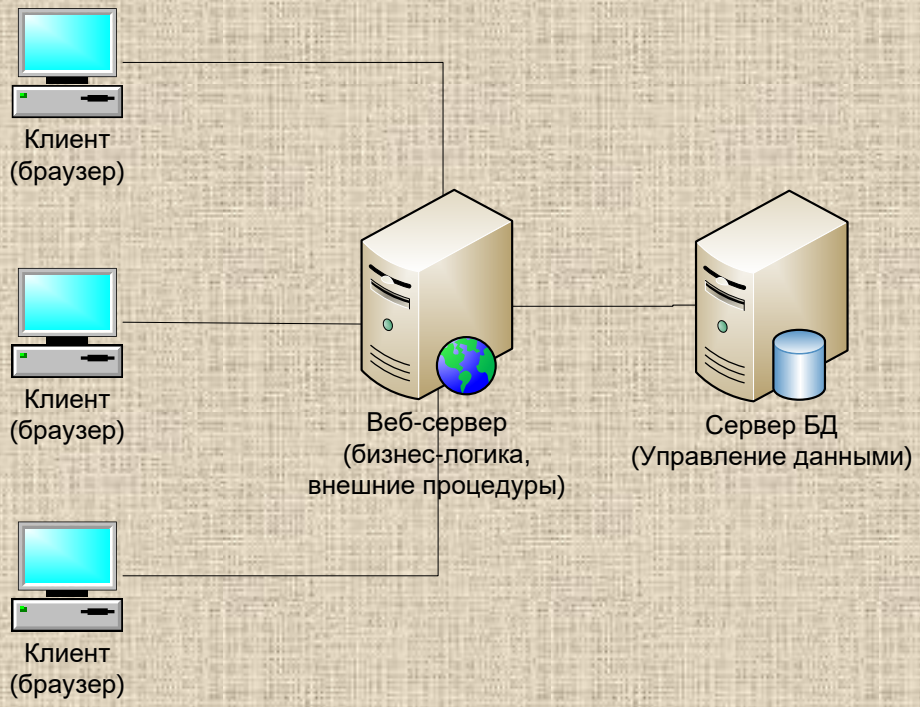
Архитектура распределенных систем с репликацией



Данными между различными рабочими станциями и централизованным хранилищем данных, передаются репликацией.

При вводе информации на рабочих станциях – данные также записываются в локальную базу данных, а лишь затем синхронизируются.

Архитектура Веб-приложений



Обычно Веб-приложения создаются как приложения в архитектуре "клиент-сервер", но серверная часть имеет различные архитектурные решения.

Особенности архитектуры Веб-приложений

- Отсутствие необходимости использовать дополнительное программное обеспечение на стороне клиента
- Возможность подключения практически неограниченного количества клиентов
- Централизованное место хранения данных
- Недоступность при отсутствии работоспособности сервера или каналов связи

Сервис-ориентированная архитектура

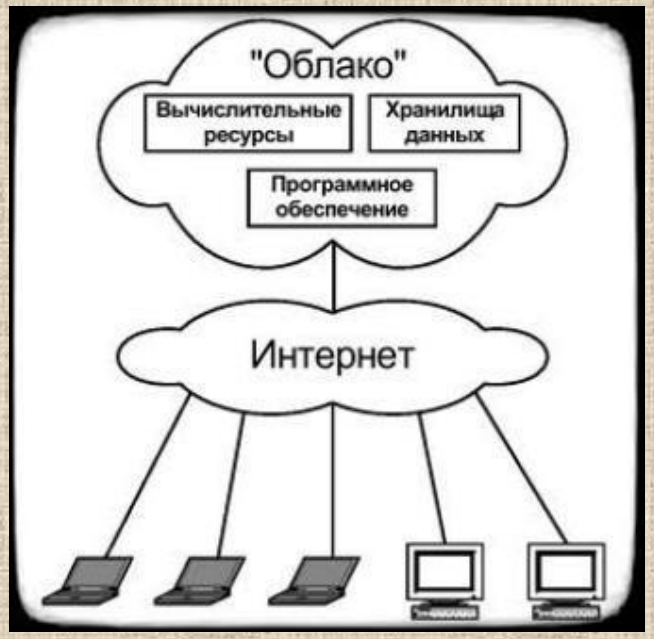
Сервис-ориентированная архитектура (SOA) – модульный подход к разработке программного обеспечения, основанный на использовании сервисов со стандартизированными интерфейсами

Принципы SOA:

- ✓ архитектура не привязана к какой-то определенной технологии
- ✓ независимость организации системы от используемой вычислительной платформы
- ✓ независимость организации системы от применяемых языков программирования
- ✓ использование сервисов, независимых от конкретных приложений, с единообразными интерфейсами доступа к ним
- ✓ организация сервисов как слабосвязанных компонентов для построения систем

Архитектура облачных вычислений

Облачные вычисления (cloud computing) — технология распределенной обработки информации, для которой используются компьютерные ресурсы и программное обеспечение, а данные предоставляются абонентам по запросу.



Виртуализация — метод системы абстрагировать вычислительные ресурсы и показывать абоненту только нужные сервисы. Виртуальная машина консолидирует физические серверы и программное обеспечение. На виртуальном сервере одновременно могут работать программы пользователей, причем под управлением разных операционных систем

Характеристики облачной архитектуры

- **Масштабируемость** — масштабируемое приложение разрешает выдержать большую нагрузку с помощью увеличения одновременно запущенных экземпляров облачных систем
- **Эластичность** — это реакция на изменения количества вычислительных ресурсов на работу информационных систем; эластичность позволяет, при необходимости, быстро подключить нужные ресурсы и их отключить
- **Мультитенантность** — архитектурное решение, разрешающее реализовывать совместную реализацию общих ресурсов
- **Плата за использование** — параметр облака, разрешающий платить за те выделяемые ресурсы, которые нужны пользователю
- **Самообслуживание** — характеристика, разрешающая пользователю лично управлять своими выделенными ресурсами

Преимущества и недостатки облачных вычислений

Преимущества архитектуры «облако»:

- ✓ удаленный доступ к информации в облаке — работать можно откуда угодно, где есть точка доступа Интернет
- ✓ снижаются требования к мощности компьютера, с которого управляется облако
- ✓ облачные технологии реализуют большую скорость обработки информации
- ✓ облачные технологии разрешают экономить на покупке, поддержке программного обеспечения и оборудования
- ✓ пользователь платит тогда, когда ему нужны эти ресурсы

Недостатки архитектуры «облако» :

- ✓ пользователь не является владельцем облака и не имеет доступ к внутренней облачной инфраструктуре
- ✓ пользователь получает тот уровень безопасности, который может дать ему провайдер
- ✓ требуется быстрый доступ в интернет

ТИПОВЫЕ РЕШЕНИЯ (ПАТТЕРНЫ)

Типовые решения

Типовые решения (шаблоны , паттерны, patterns) – это многократно применяемые архитектурные конструкции, предоставляющие решение общих проблем проектирования в рамках конкретного контекста и описывающие значимость этого решения.

Преимущества использования типовых решений

- ✓ Документируют простые механизмы, которые работают.
- ✓ Обеспечивают общий словарь для архитекторов и программистов
- ✓ Позволяют описают архитектурные решения сжато (лаконично) с использованием комбинации паттернов
- ✓ Позволяют повторно использовать архитектурные, проектные решения и решения по реализации

Кристофер Александер:

"Каждое типовое решение описывает некую повторяющуюся проблему и ключ к ее разгадке, причем таким образом, что вы можете пользоваться этим ключом многократно, ни разу не придя к одному и тому же результату"

История шаблонов в программном обеспечении

1987 год Кент Бэк и Вард Каннигем шаблоны применительно к разработке программного обеспечения для разработки графических оболочек на языке Smalltalk.
1988 год Эрих Гамма начал писать докторскую диссертацию при Цюрихском университете об общей переносимости этой методики на разработку программ.
1991 год Джеймс Коплин Advanced C++ Idioms.

1991 год Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидс Design Patterns – ***Elements of Reusable Object-Oriented Software*** (23 шаблона проектирования). Команда авторов известна общественности под названием «Банда четырех» (***Gang of Four, часто сокращается до GoF***). Именно эта книга стала причиной роста популярности шаблонов проектирования.

Мартин Фаулер
Шаблоны проектирования корпоративных программных приложений (Patterns of Enterprise Application Architecture - PoEAA)

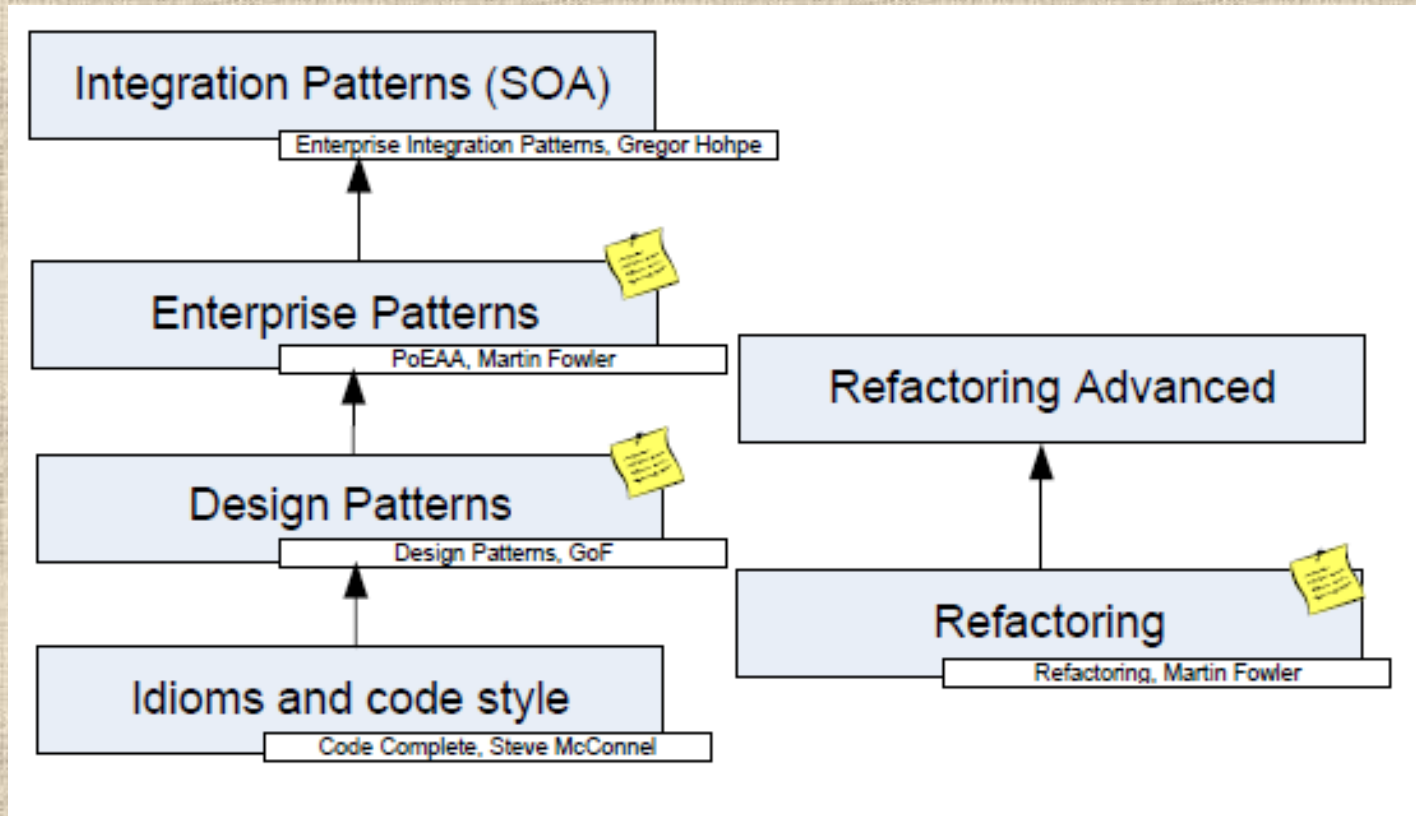
Структура типового решения

1. **Название паттерна**. Представляет собой уникальное смысловое имя, однозначно определяющее данную задачу или проблему и ее решение.
2. **Решаемая задача**. Здесь дается понимание того, почему решаемая проблема действительно является таковой, четко описывает ее границы.
3. **Решение, принцип работы.** Здесь указывается, как именно данное решение связано с проблемой, приводится пути ее решения.
4. **Результаты использования паттерна**. Обычно приводятся достоинства, недостатки и компромиссы.

Ценность типового решения для проектировщика заключается не в новизне идеи;
главное— это помощь в описании и сообщении этой идеи

Классификация типовых решений

1. Интеграционные шаблоны
2. Архитектурные паттерны
3. Паттерны проектирования
4. Идиомы программирования
5. Шаблоны рефакторинга



«Расслоение» системы

Представление (presentation)

Предоставление услуг, отображение данных, обработка событий пользовательского интерфейса (щелчков кнопками мыши и нажатий клавиш), обслуживание запросов HTTP, поддержка функций командной строки и API пакетного выполнения

Домен (Domain)

Бизнес-логика приложения

Источник данных (Data Source)

Обращение к базе данных, обмен сообщениями, управление транзакциями и т.д.

Не путать слои (layers) и уровни (tiers)!!!

Преимущества «расслоения» системы

- ✓ Каждый слой можно воспринимать как единое целое, не особенно заботясь о наличии других слоев
- ✓ Можно выбирать альтернативную реализацию базовых слоев, без изменения всей системы
- ✓ Зависимость между слоями можно свести к минимуму
- ✓ Каждый слой является удачным кандидатом на стандартизацию
- ✓ Созданный слой может служить основой для нескольких различных реализаций слоев более высокого уровня

Недостатки «расслоения» системы

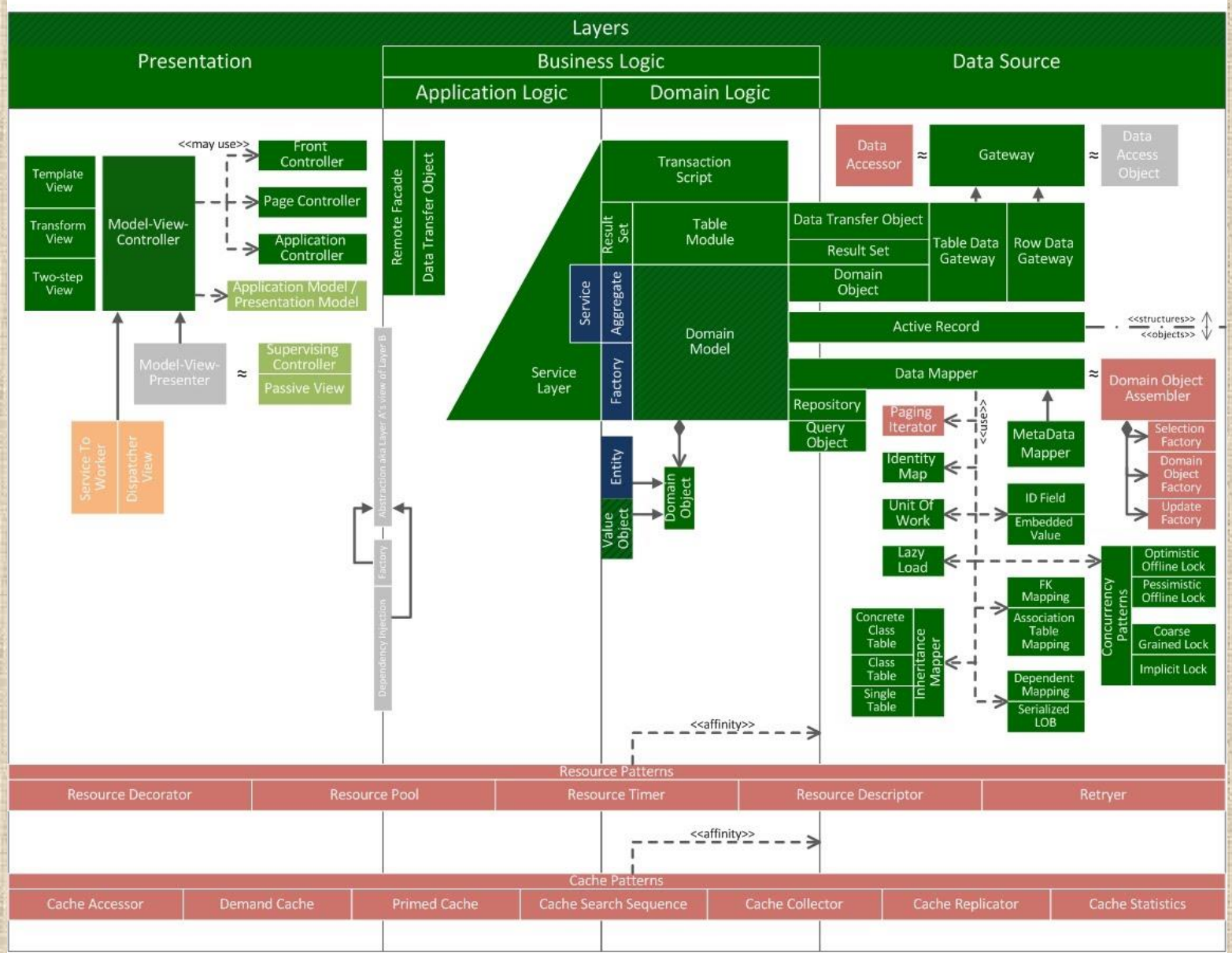
- Слои способны удачно инкапсулировать многое, но не все: модификация одного слоя подчас связана с необходимостью внесения каскадных изменений в остальные слои.

- Наличие избыточных слоев нередко снижает производительность системы. При переходе от слоя к слою моделируемые сущности обычно подвергаются преобразованиям из одного представления в другое.

Самое трудное при использовании архитектурных слоев — это определение содержимого и границ ответственности каждого слоя.

Enterprise Design Patterns

ekr@list.ru



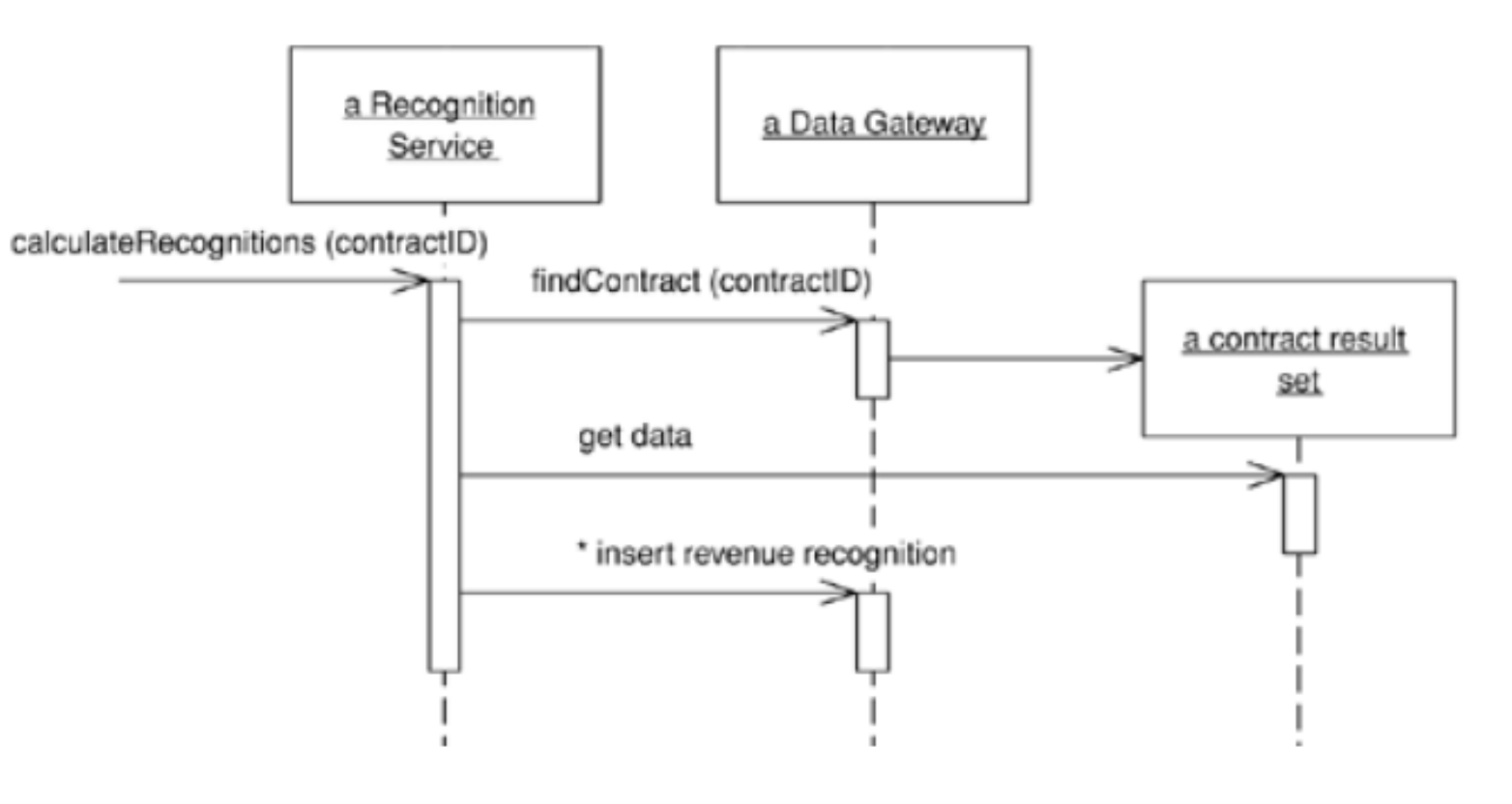
Проблема

Необходимость изменения исходного кода реализации бизнес логики при расширении/изменении функциональности приложения.

Типовые решения для представления бизнес-логики

1. Сценарий транзакции (Transaction Script)
2. Модель предметной области (Domain Model)
3. Модуль таблицы (Table Module)
4. Уровень служб (Service Level)

Типовое решение «Сценарий транзакции»
сценарий действия, бизнес-транзакция



Бизнес-логика описывается набором процедур, по одной на каждую (составную) операцию, которую способно выполнять приложение.

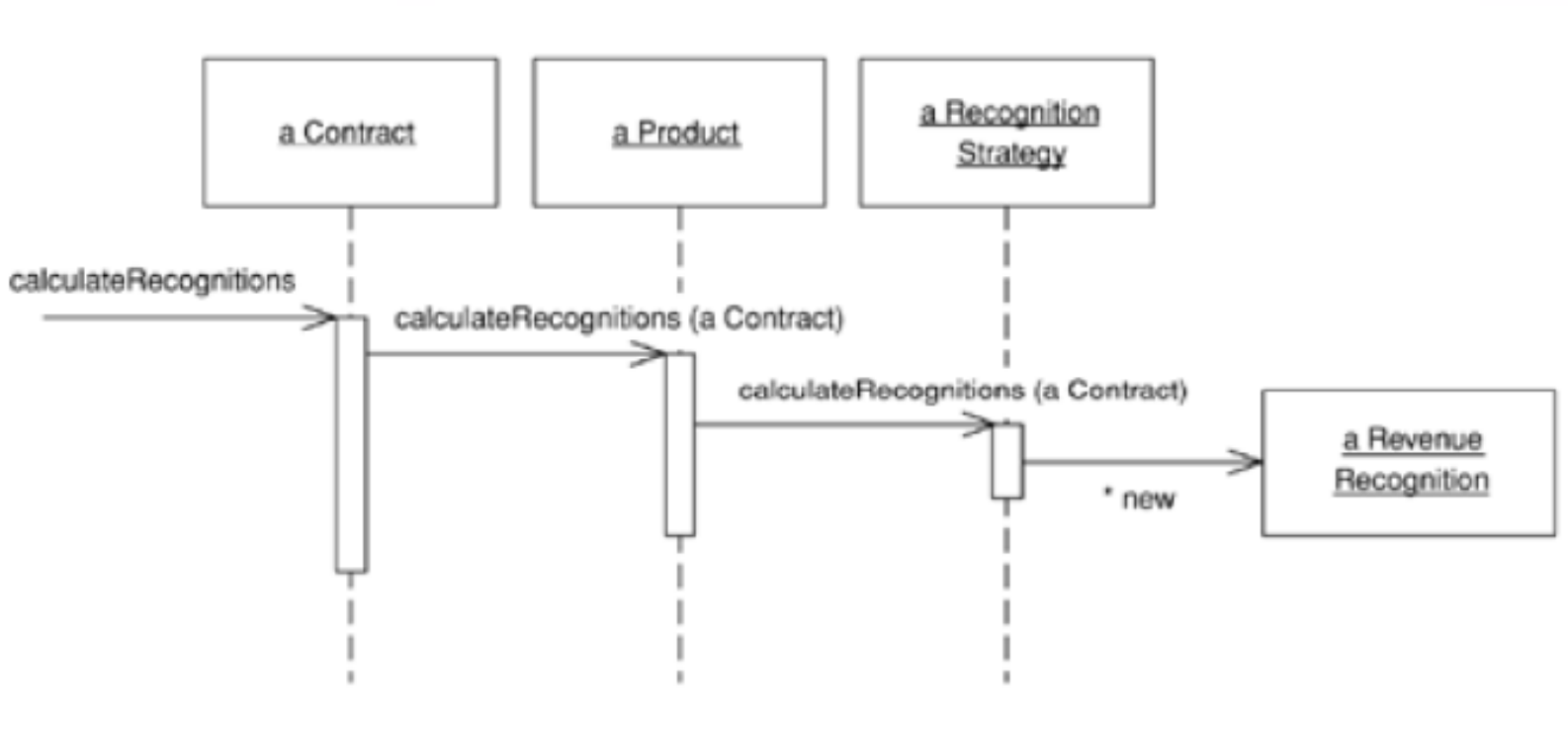
Сценарий транзакции

Как это работает:
Процедура, которая получает на вход информацию от слоя представления, обрабатывает ее, проводя необходимые проверки и вычисления, сохраняет в базе данных и активизирует операции других систем. Затем процедура возвращает слою представления определенные данные, возможно, осуществляя вспомогательные операции для форматирования содержимого результата.

Преимущества:
- ✓ представляет удобную процедурную модель, легко воспринимаемую разработчиками
- ✓ удачно сочетается с простыми схемами организации слоя источника данных
- ✓ определяет четкие границы транзакции

Недостатки:
- ➢ при возрастании сложности бизнес-логики появляются транзакции с похожим поведением, что может привести к дублированию исходного кода
- ➢ приложение может превратиться в «свалку» функций без четко выраженной структуры

Типовое решение «Модель предметной области»
объекты инкапсулируют данные и функции



Переход к объектной парадигме: вместо использования одной подпрограммы, несущей в себе всю логику, которая соответствует некоторому действию пользователя, каждый объект наделяется только функциями, отвечающими его природе. Для каждого экземпляра сущности создается отдельный объект в программе.

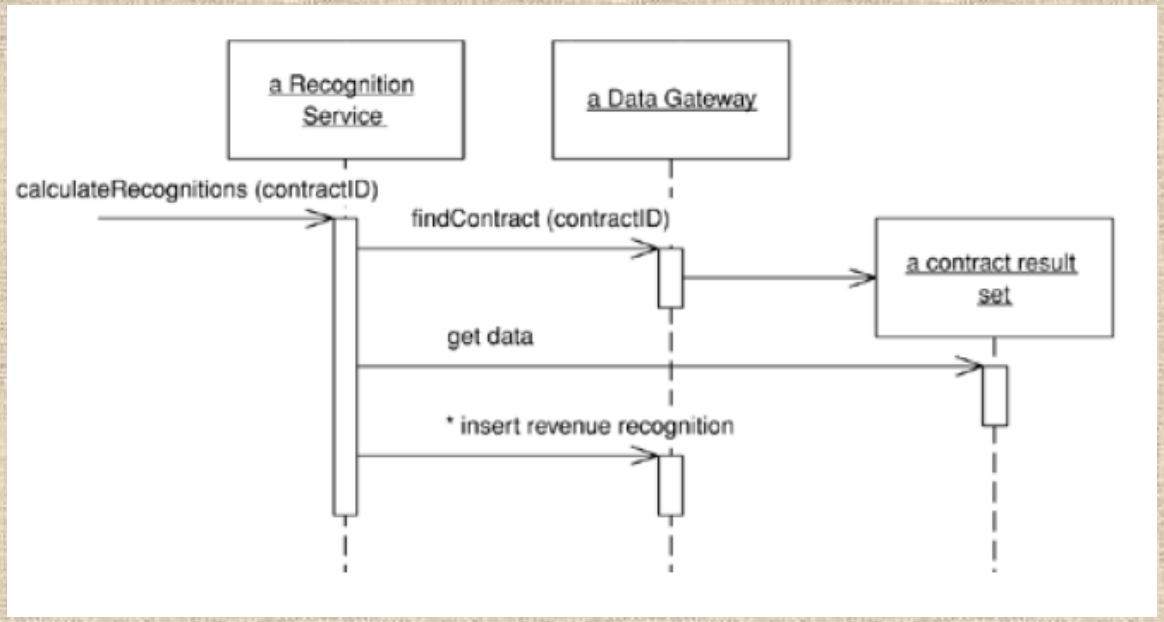
Модель предметной области

Как это работает:
Операция представляется взаимодействием нескольких объектов, каждый из которых реализует присущую только ему часть ответственности

Преимущества:
- ✓ предоставляет возможности преодоления сложности бизнес-логики
- ✓ схема организации бизнес-логики соответствует структуре и семантики предметной области и облегчает взаимодействия разработчиков, аналитиков, пользователей

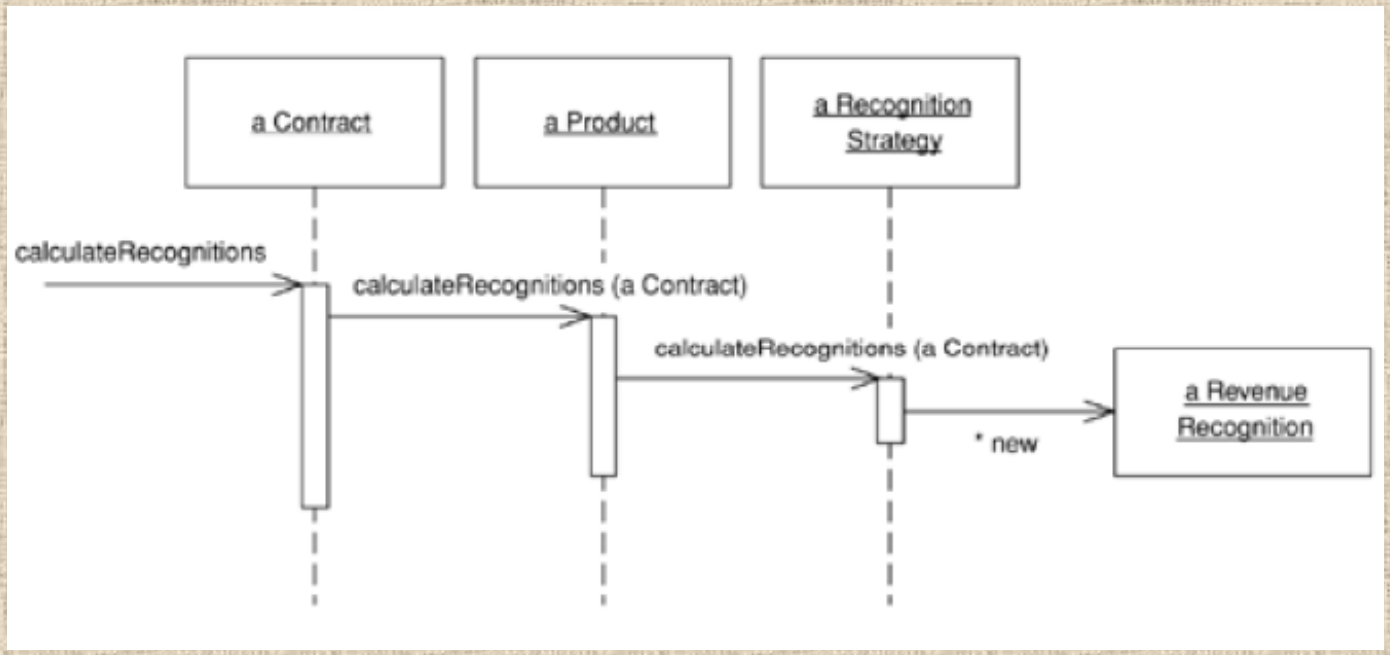
Недостатки:
- ➢ сложность освоения
- ➢ чем сложнее предметная область, тем сложнее реализовать преобразование объектной модели в реляционную структуру хранения в базе данных

Сравнение типовых решений

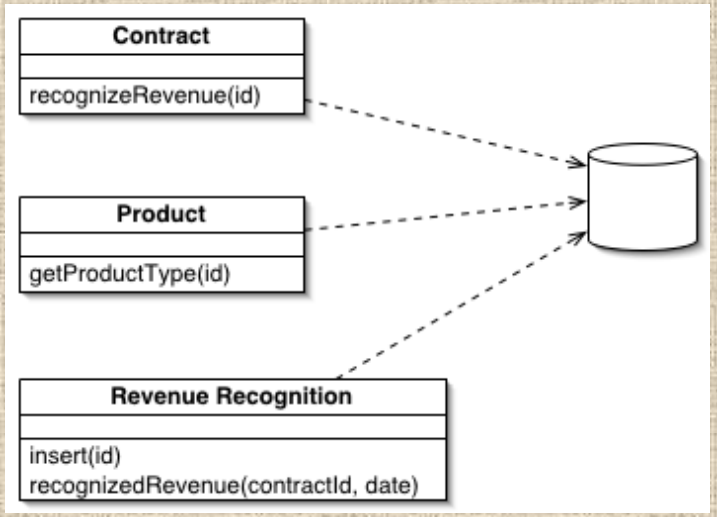


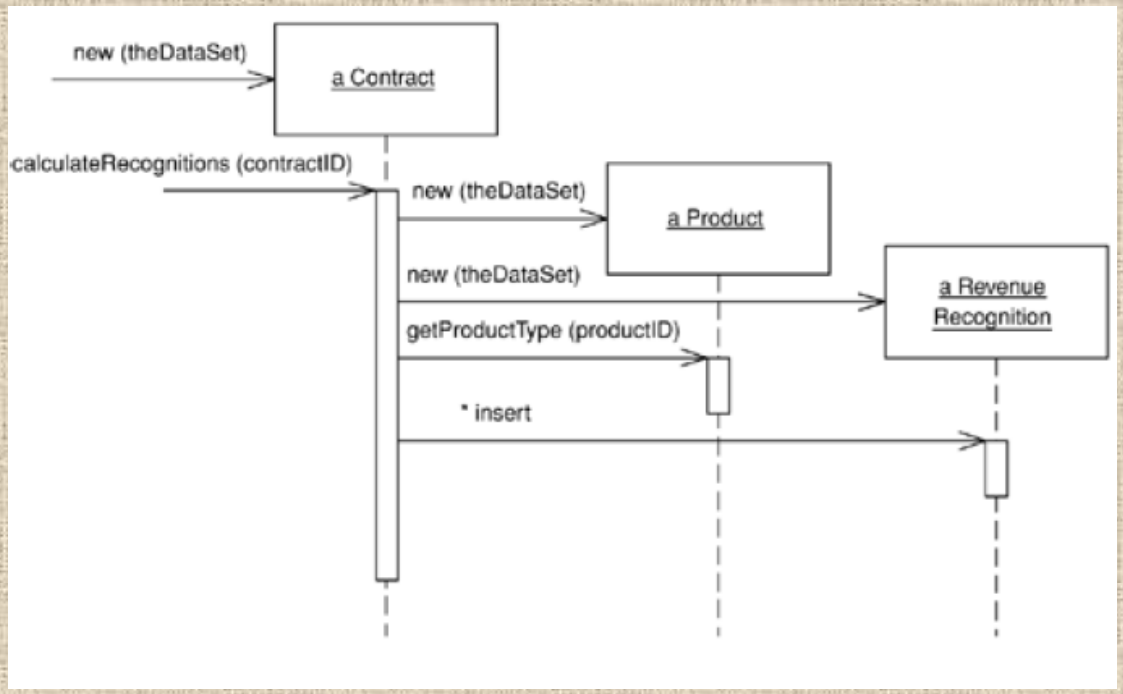
«Сценарий транзакции»

«Модель предметной области»



Типовое решение «Модуль таблицы»
один объект представляет множество других объектов





Один объект обрабатывает бизнес-запросы для всех строк таблицы или представления базы данных.
Является промежуточным вариантом реализации между типовым решением «Сценарий транзакции» и «Модель предметной области»

Модуль таблицы

Как это работает

Модуль таблицы применяется совместно с типовым решением «Множество записей» (Record Set). Посылая запросы к базе данных, пользователь прежде всего формирует объект множество записей, а затем создает объект (на картинке Контракт), передавая ему множество записей в качестве аргумента. Если потребуется выполнять операции над отдельным объектом, следует сообщить объекту соответствующий идентификатор (на рисунке – contractID).

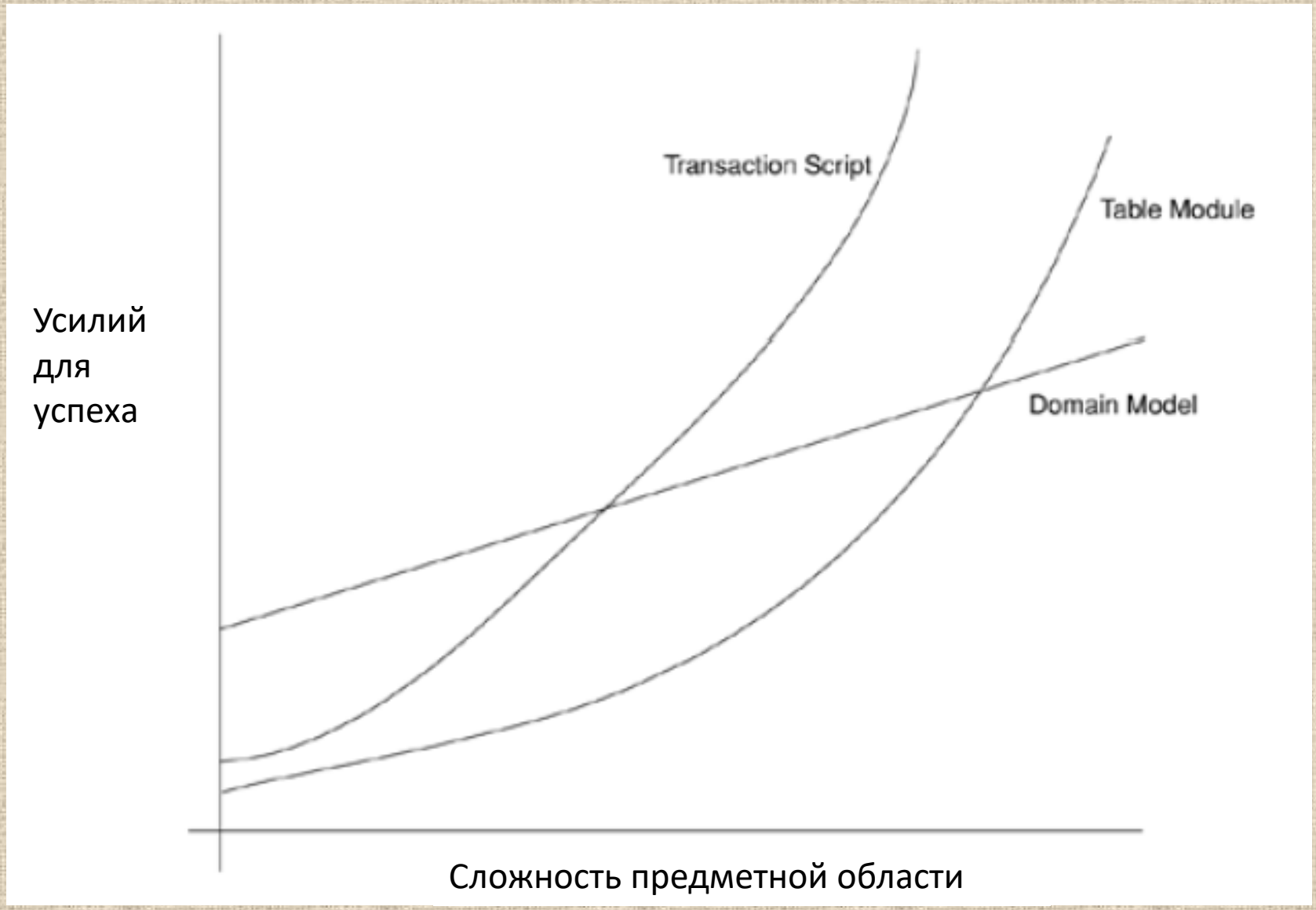
Преимущества:

- ✓ данное решение хорошо сочетается с остальными аспектами архитектуры, например, с графическими интерфейсами пользователя, работающими с результатами запросов, организованными в виде множества записей

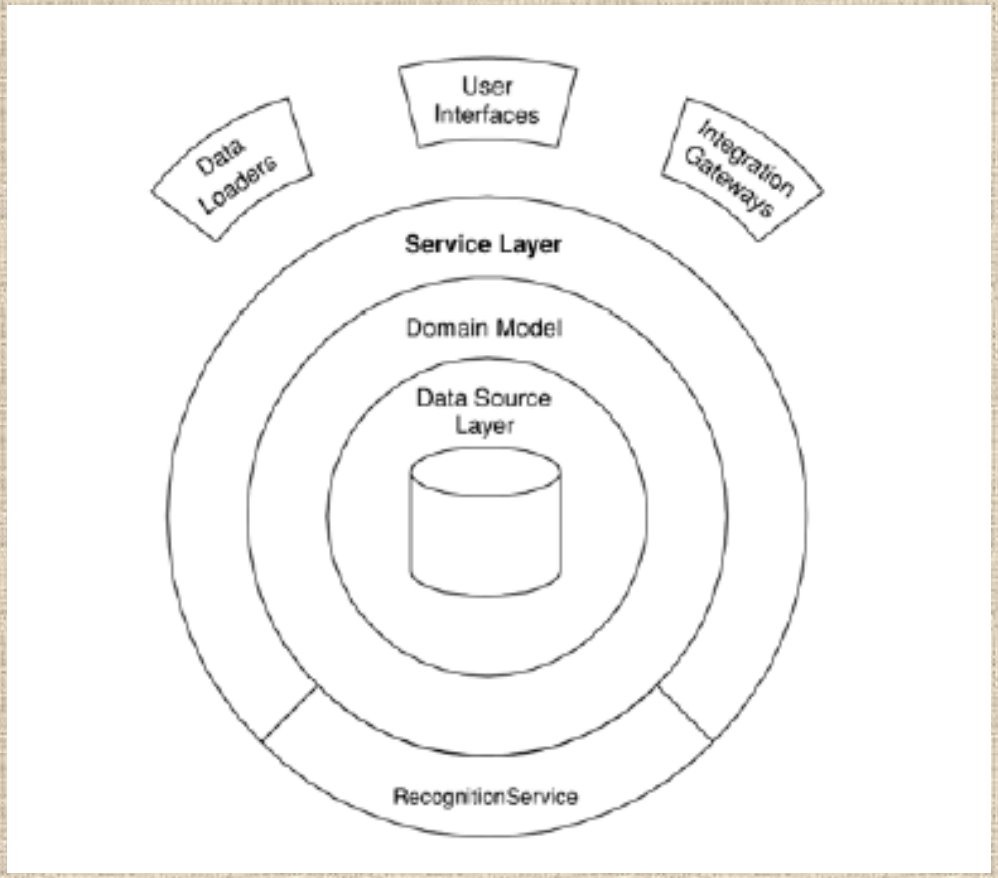
Недостатки:

- ➢ не позволяет использовать многие технологии объектного программирования и объектно-ориентированные типовые решения (например, наследование)

Какое решение выбрать?



Типовое решение «Уровень служб» (Service Level)



Варианты реализации:

1. Промежуточный интерфейс, который перенаправляет клиентские вызовы соответствующим объектам

2. В рамках слоя служб представляет большую часть логики в виде сценариев транзакции

3. Модель «контроллер-сущность» (контроллеры вариантов использования, объекты – сущности)

Располагается «поверх» типовых решений «Сценарий транзакции» и « Модель предметной области», определяя границы и поддерживания интерфейс приложения.

Уровень служб

Преимущества:

- ✓ подходит для размещения логики управления транзакциями и обеспечения безопасности
- ✓ определяет общий набор интерфейсов приложения для различных видов клиентов

Недостатки:

- ➢ возможна «перегрузка» уровня служб бизнес-логикой

Рекомендация:

Делать уровень служб только при необходимости и стремиться к его «тонкости»

Проблемы работы с данными (ресурсами)

Изменение кода приложения при изменении структуры хранения или физической реализации базы данных (ресурсов).

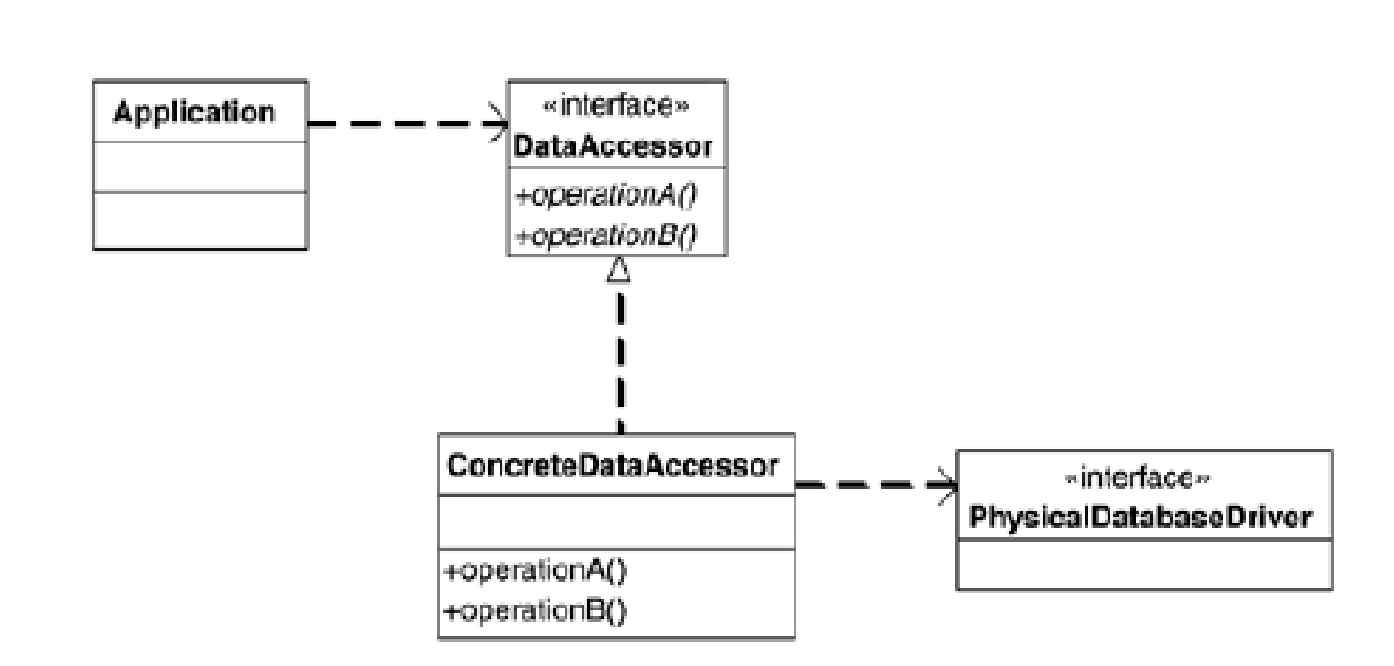
Синхронизация доступа к базе данных (ресурсам), обеспечение совместной работы с данными (ресурсами) нескольких пользователей.

Типовые решения для работы с данными

Источники данных	Ресурсные паттерны	Паттерны ввода-вывода	Паттерны кэширования
Получатель данных (Data Accessor)	Декоратор ресурса (Resource Decorator)	Компоновщик объекта предметной области (Domain Object Assembler)	Доступ к кэш (Cache Accessor)
Шлюз таблицы данных (Table Data Gateway)	Ресурсный пул (Resource Pool)	Фабрика выбора (Selection Factory)	Кэш по требованию (Demand Cache)
Шлюз записи данных (Row Data Gateway)	Ресурсный таймер (Resource Timer)	Фабрика объекта предметной области (Domain Object Factory)	Кэш с предзагрузкой (Primed Cache)
Активная запись (Active Record)	Дескриптор ресурса (Resource Descriptor)	Фабрика изменения (Update Factory)	Кэш с последовательным поиском (Cache Search Sequence)
Преобразователь данных (Data Mapper)	Повторитель (Retrier)	Постраничный итератор (Paging Iterator)	Коллектор (очиститель) кэша (Cache Collector)
			Репликатор кэш (Cache Replicator)
			Статистика кэша (Cache Statistics)

Типовое решение «Получатель данных» (Data Accessor)

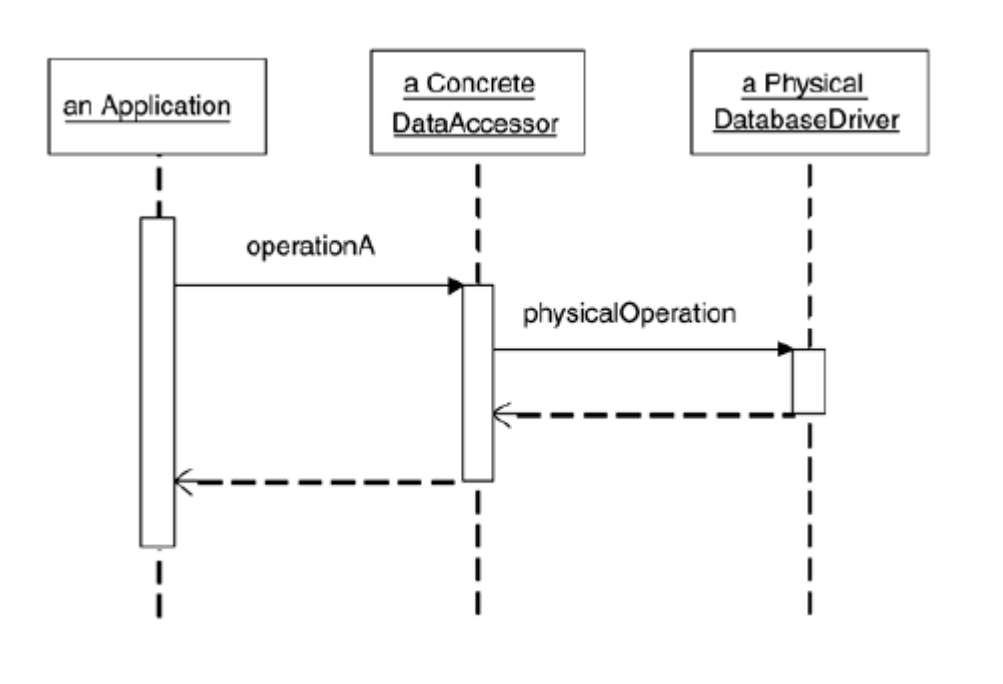
отделяет логические операции доступа к данным от операций получения данных низкого уровня



Приложение знает о допустимых операциях работы с данными, но освобождено от ответственности за получение данных из конкретного источника.

«Получатель данных»

Как это работает



Data Accessor инкапсулирует все операции, необходимые для физического получения данных из конкретного источника с учетом особенностей платформы хранения, скрывая их от прикладного приложения. Одна логическая операция может реализовываться по разному на физическом уровне доступа к данным в зависимости от платформы.

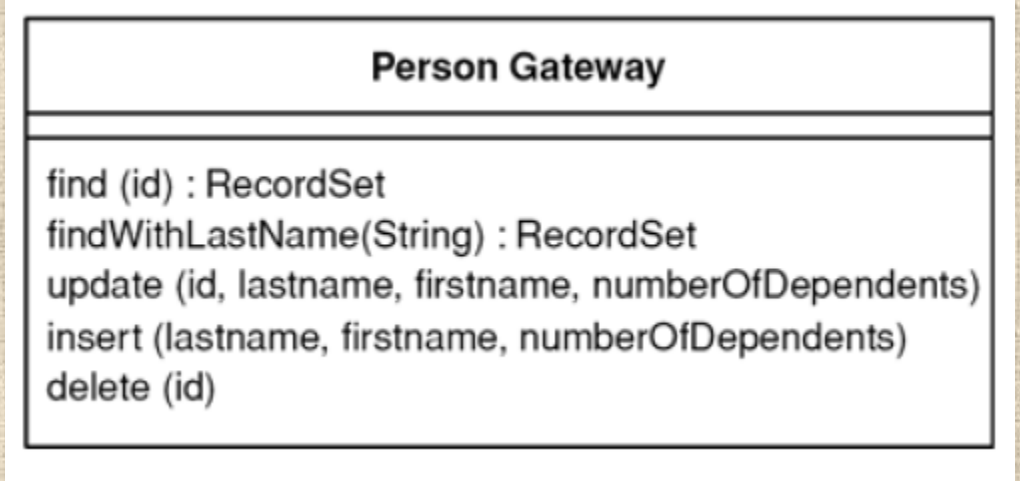
«Получатель данных»

Преимущества:

- ✓ «чистый» код приложения: с помощью хорошо продуманного и организованного кода доступа к данным приложение можно сделать более «чистым» и сопровождаемым
- ✓ простота внедрения новых платформ хранения данных и особенностей операций работы с источниками данными
- ✓ возможности оптимизации кода работы с источниками данных без изменения всего приложения
- ✓ возможность одновременной поддержки нескольких реализаций физического уровня хранения данных и переключения между ними

Типовое решение «Шлюз таблицы данных» (Data Table Gateway)

один объект обрабатывает все записи таблицы



Содержит в себе все команды SQL, необходимые для извлечения, вставки, обновления и удаления данных из таблицы или представления. Методы этого типового решения используются другими объектами для взаимодействия с базой данных.

Шлюз таблицы данных

Как это работает

Содержит несколько методов поиска данных в таблице.

Возвращает результаты поиска данных в таблицы следующими способами:

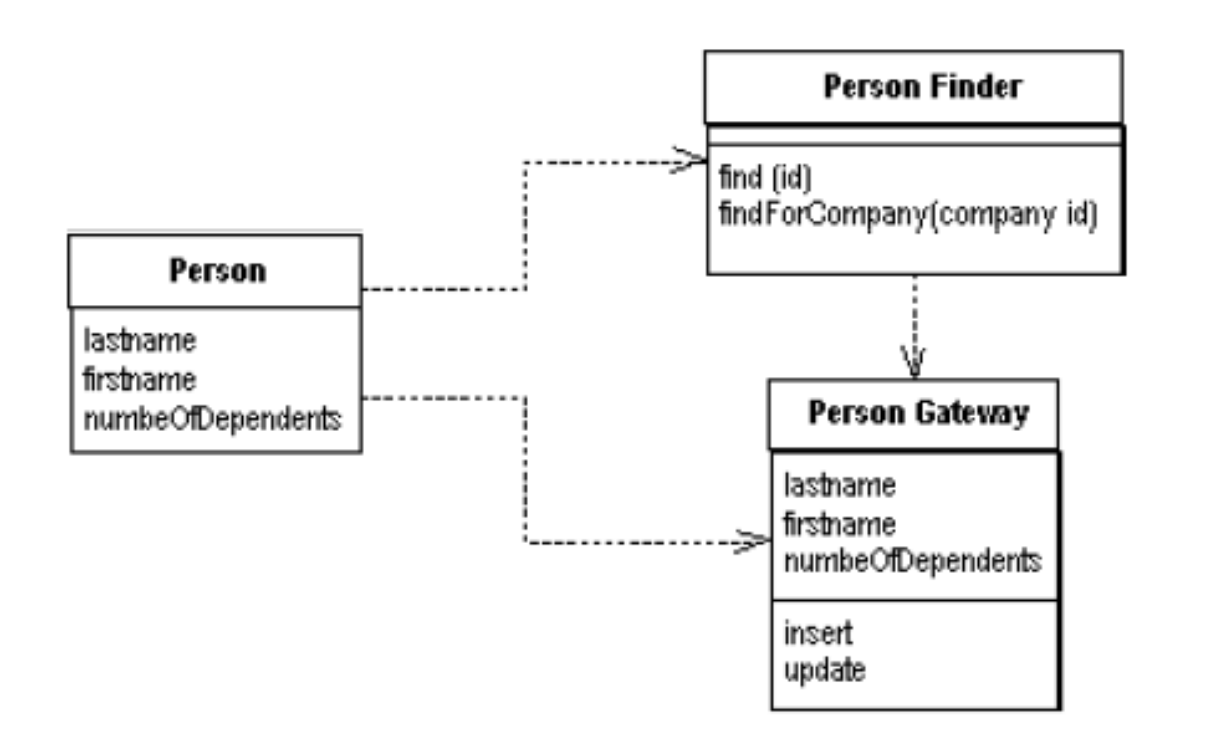
- Коллекция (Collection)
- Объект передачи данных (Data Transfer Object)
- Множество записей (Record Set)
- Объект предметной области (Domain Object)

Преимущества:

- ✓ наиболее простое типовое решение для работы с данными
- ✓ подходит практически для любой платформы, так как по сути представляет собой оболочку sql команд работы с данными
- ✓ единый интерфейс как для работы с данными, так и для работы с хранимыми процедурами в СУБД

Типовое решение «Шлюз записи данных» (Row Data Gateway)

один объект на каждую запись таблицы



Типовое решение «шлюз записи данных» предоставляет объекты, которые полностью аналогичны записям базы данных, однако могут быть доступны с помощью обычных механизмов используемого языка программирования. Все детали доступа к источнику данных скрыты за интерфейсом.

Шлюз записи данных

Как это работает

Шлюз записи данных выступает в роли объекта, полностью повторяющего одну запись. Каждому столбцу таблицы соответствует поле объекта. Обычно шлюз записи данных должен выполнять все возможные преобразования типов источника данных в типы, используемые приложением, однако эти преобразования весьма просты.

Рассматриваемое типовое решение содержит все данные о строке, поэтому клиент имеет возможность непосредственного доступа к шлюзу записи данных. Шлюз содержит только простые операции вставки, изменения или удаления записи, без какой-либо бизнес-логики.

Реализация методов поиска, генерирующих экземпляры шлюзов записей данных:

-использование статических методов (недостаток – невозможность полиморфизма, что могло бы пригодиться, если понадобится определить разные методы поиска для различных источников данных);

-создание отдельных объектов поиска для каждой таблицы реляционной базы данных.

Преимущества:

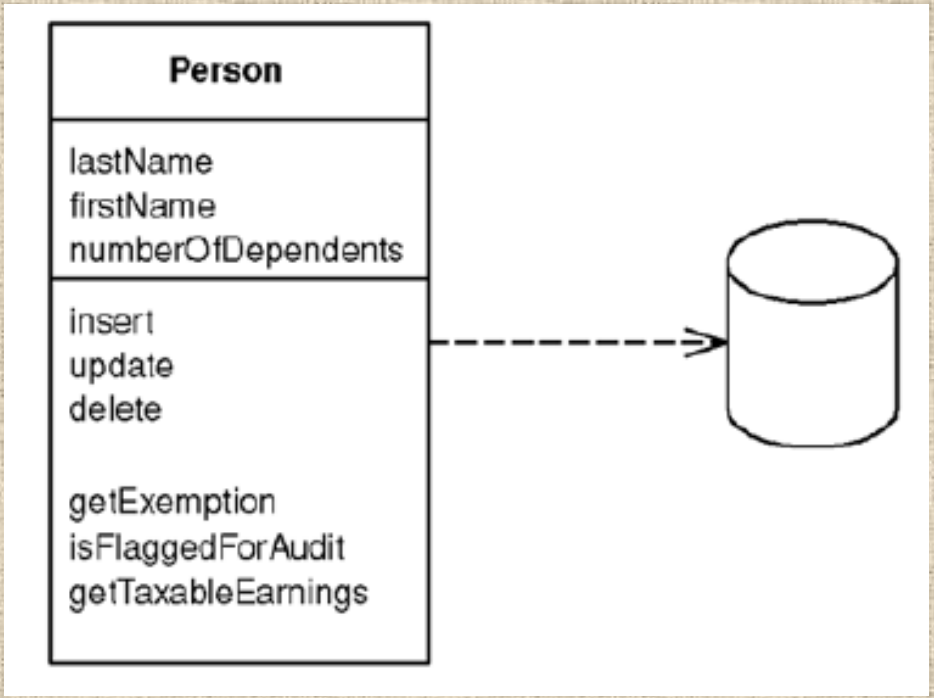
- ✓ наиболее простое типовое решение для работы с отдельными записями таблиц
- ✓ возможно использование для сокрытия структуры базы данных от объектов домена (предметной области)

Недостаток:

- ✓ трудоемкая реализация для крупных систем
- ✓ наличие трех представлений данных: в доменных объектах, в шлюзах записей и в таблицах базы данных

Типовое решение «Активная запись» (Active Record)

объект представляет запись таблицы базы данных, инкапсулирует операции доступа к базе данных и добавляет к данным логику домена



Объект охватывает и данные и поведение.
Большая часть его данных является постоянной и должна храниться в базе данных. Логика доступа к данным включается в объект домена. В этом случае все знают, как считывать данные из базы данных и как их записывать в нее.

Активная запись

Как это работает

В основе типового решения активная запись лежит модель предметной области (Domain Model), классы которой повторяют структуру записей используемой базы данных. Каждая активная запись отвечает за сохранение и загрузку информации в базу данных, а также за логику домена, применяемую к данным. Это может быть вся бизнес-логика приложения.

Структура данных активной записи должна в точности соответствовать таковой в таблице базы данных: каждое поле объекта должно соответствовать одному столбцу таблицы. Значения полей следует оставлять такими же, какими они были получены в результате выполнения SQL-команд.

Как правило, типовое решение активная запись включает в себя методы, предназначенные для выполнения следующих операций:

- создание экземпляра активной записи на основе строки, полученной в результате выполнения SQL-запроса;
- создание нового экземпляра активной записи для последующей вставки в таблицу;
- статические методы поиска, выполняющие стандартные SQL-запросы и возвращающие активные записи;
- обновление базы данных и вставка в нее данных из активной записи;
- извлечение и установка значений полей (get- и set-методы);
- реализация некоторых фрагментов бизнес-логики.

Активная запись

Преимущества:

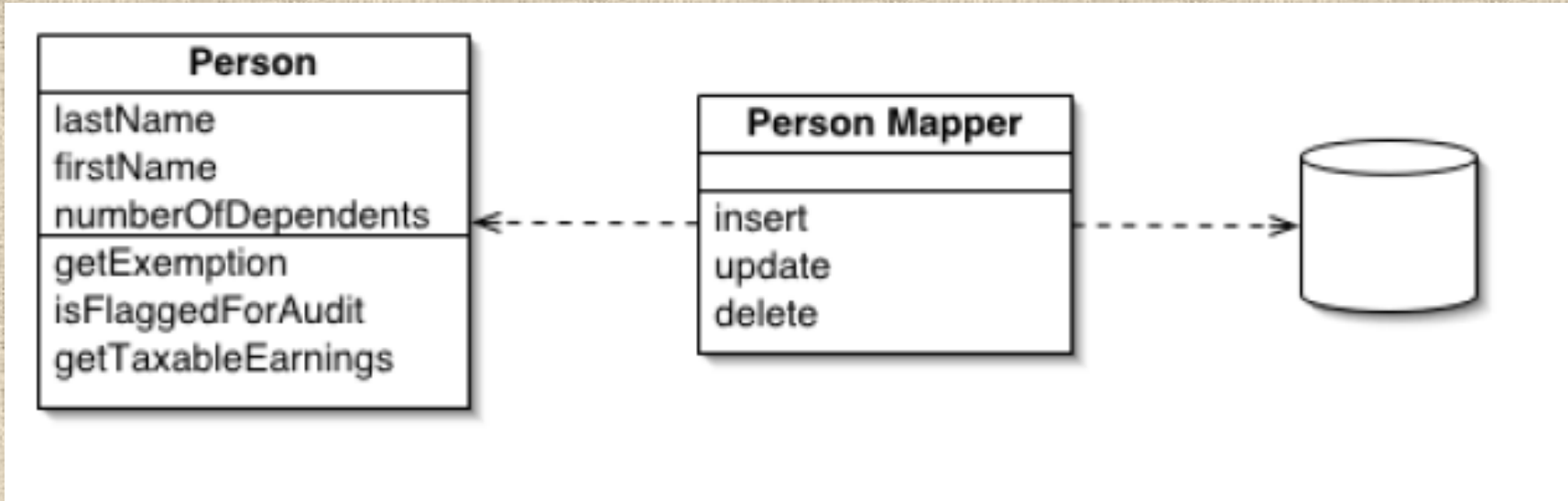
- ✓ очень удобны для разработчиков, так как сочетают структуру хранения и операции бизнес-логики
- ✓ в отличии от шлюза записи данных, статические методы поиска не выделяются в отдельный класс, так как активная запись тесно связана с конкретной реализацией базы данных
- ✓ можно применять не только к таблицам, но и к представлениям
- ✓ хорошо справляется с извлечением и проверкой на правильность отдельной записи
- ✓ простота реализации
- ✓ хорошо сочетается с типовым решением «сценарий транзакции» для инкапсуляции повторяющейся несложной бизнес логики

Недостаток:

- ✓ подходит для реализации только не очень сложной бизнес-логики
- ✓ хороши только тогда, когда точно отображаются на таблицы базы данных
- ✓ Неудобны для создания коллекций, использования наследования и других отношений, так как добавление «по кускам» приведет к неразберихе
- ✓ тесная зависимость структуры объектов от структуры базы данных, трудность изменения структуры базы данных и структуры активной записи

Типовое решение «Преобразователь данных» (Data Mapper)

осуществляет передачу данных между объектами и базой данных, сохраняя последние независимыми друг от друга и от самого преобразования



Объектная модель и реляционная СУБД должны обмениваться данными. Несовпадение схем делает эту задачу крайне сложной. Если объект "знает" о структуре реляционной базы данных, изменение одного из них приводит к необходимости изменения другого.

Типовое решение преобразователь данных представляет собой слой программного обеспечения, которое отделяет объекты, расположенные в оперативной памяти, от базы данных. В функции преобразователя данных входит передача данных между объектами и базой данных и изоляция их друг от друга.

Преобразователь данных

Как это работает

Основной функцией преобразователя данных является отделение домена от источника данных.

Преобразователи должны уметь обрабатывать классы, поля которых объединяются одной таблицей, классы, соответствующие нескольким таблицам, классы с наследованием, а также справляться со связыванием загруженных объектов.

Обычно запрос клиента приводит к загрузке целого графа связанных между собой объектов. В этом случае разработчик преобразователя должен решить, как много объектов можно загрузить за один раз. Поскольку число обращений к базе данных должно быть как можно меньшим, методам поиска должно быть известно, как клиенты используют объекты, чтобы определить оптимальное количество загружаемых данных.

Как правило, объекты тесно связаны между собой, поэтому на каком-то этапе загрузку данных следует прерывать. В противном случае выполнение одного запроса может привести к загрузке всей базы данных!

В приложении может быть один или несколько преобразователей данных.

Рекомендуется создать по одному преобразователю для каждого класса домена или для корневого класса в иерархии доменов.

Преобразователи должны иметь доступ к полям объектов домена. Рекомендуемый вариант – использовать специальные get- и set- методы, снабженные метками, привязывающими их к преобразователям данных.

Универсальные преобразователи работают на основе метаданных, описывающих соответствие объектов домена и таблицы базы данных.

Преобразователь данных

Преимущества:

- ✓ обеспечивается возможность независимого изменения схемы базы данных и объектной модели приложения (модели предметной области)
- ✓ возможность работы с моделью предметной областью без учета структуры базы данных как в процессе проектирования, так и во время сборки и тестирования проекта
- ✓ применение преобразователей данных помогает в написании кода, поскольку позволяет работать с объектами домена без необходимости понимать принцип хранения соответствующей информации в базе данных
- ✓ существует большое количество готовых преобразователей данных, которые могут использоваться при разработке нового приложения

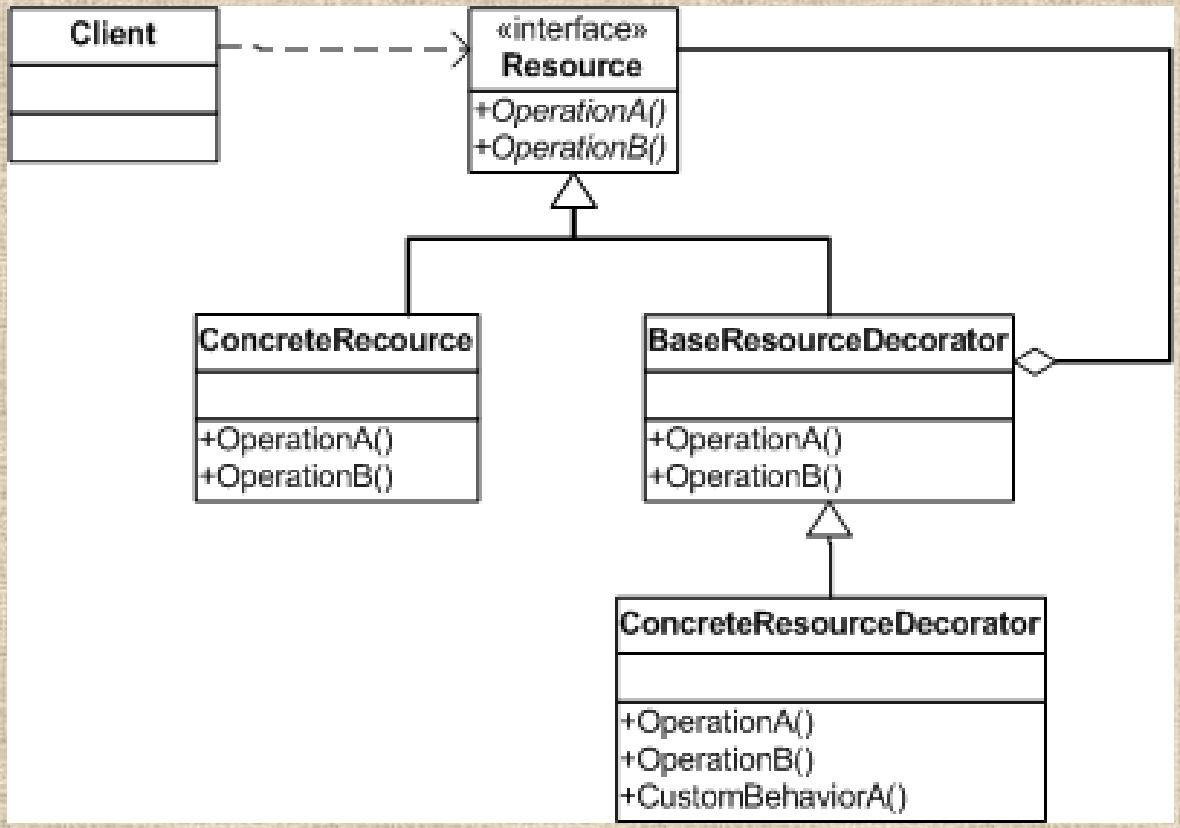
Недостаток:

- ✓ необходимость реализации дополнительного слоя кода

Ресурсные паттерны

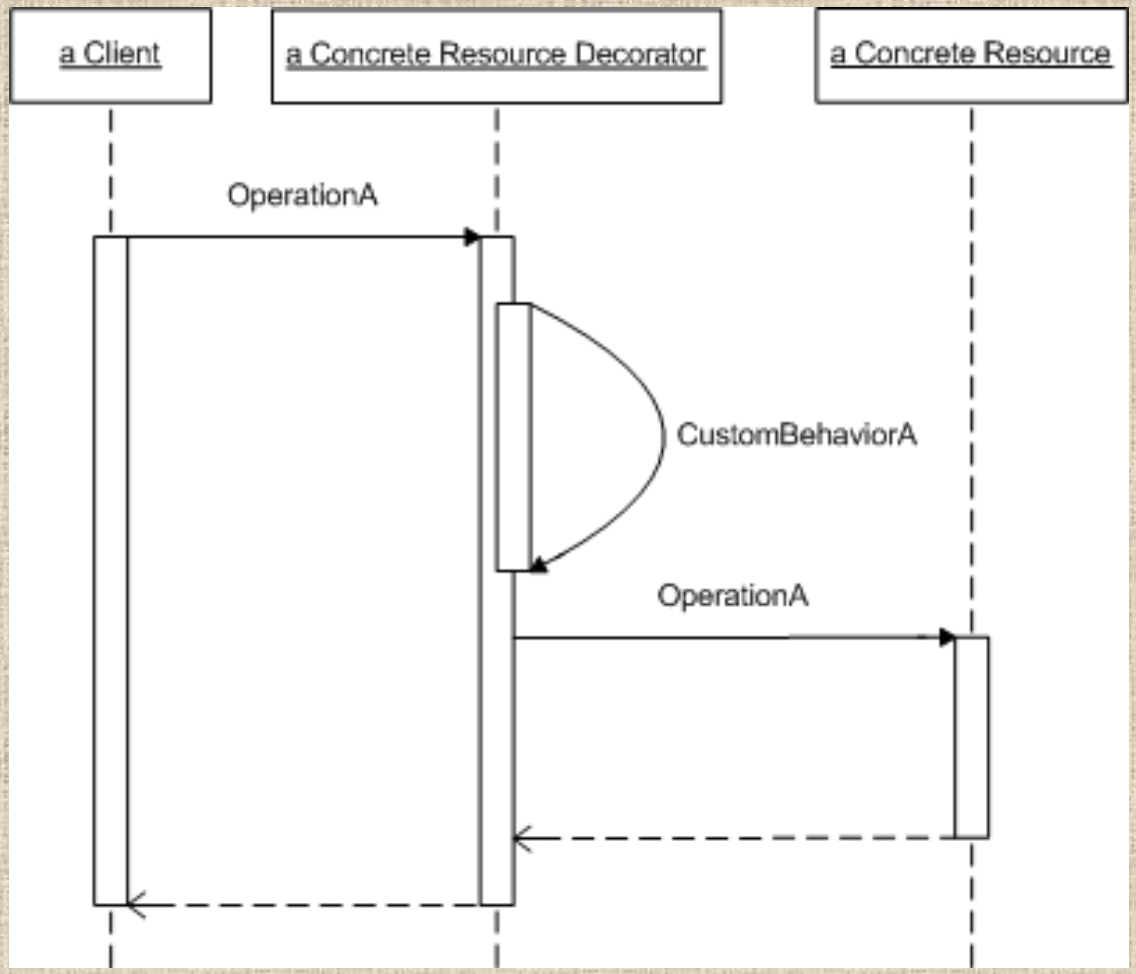
- Декоратор ресурса (Resource Decorator)
- Ресурсный пул (Resource Pool)
- Ресурсный таймер (Resource Timer)
- Дескриптор ресурса (Resource Descriptor)
- Повторитель (Retryer)

Типовое решение «Декоратор ресурса» (Resource Decorator)



Динамически подключает дополнительное поведение к существующему ресурсу с минимальными изменениями для кода основного приложения. Декоратор ресурса позволяет расширить функциональность ресурса без создания подклассов или изменения его реализации.

Декоратор ресурса



Пример использования: добавление функций протоколирования и журналирования к стандартным драйверам работы с СУБД.

Декоратор ресурса

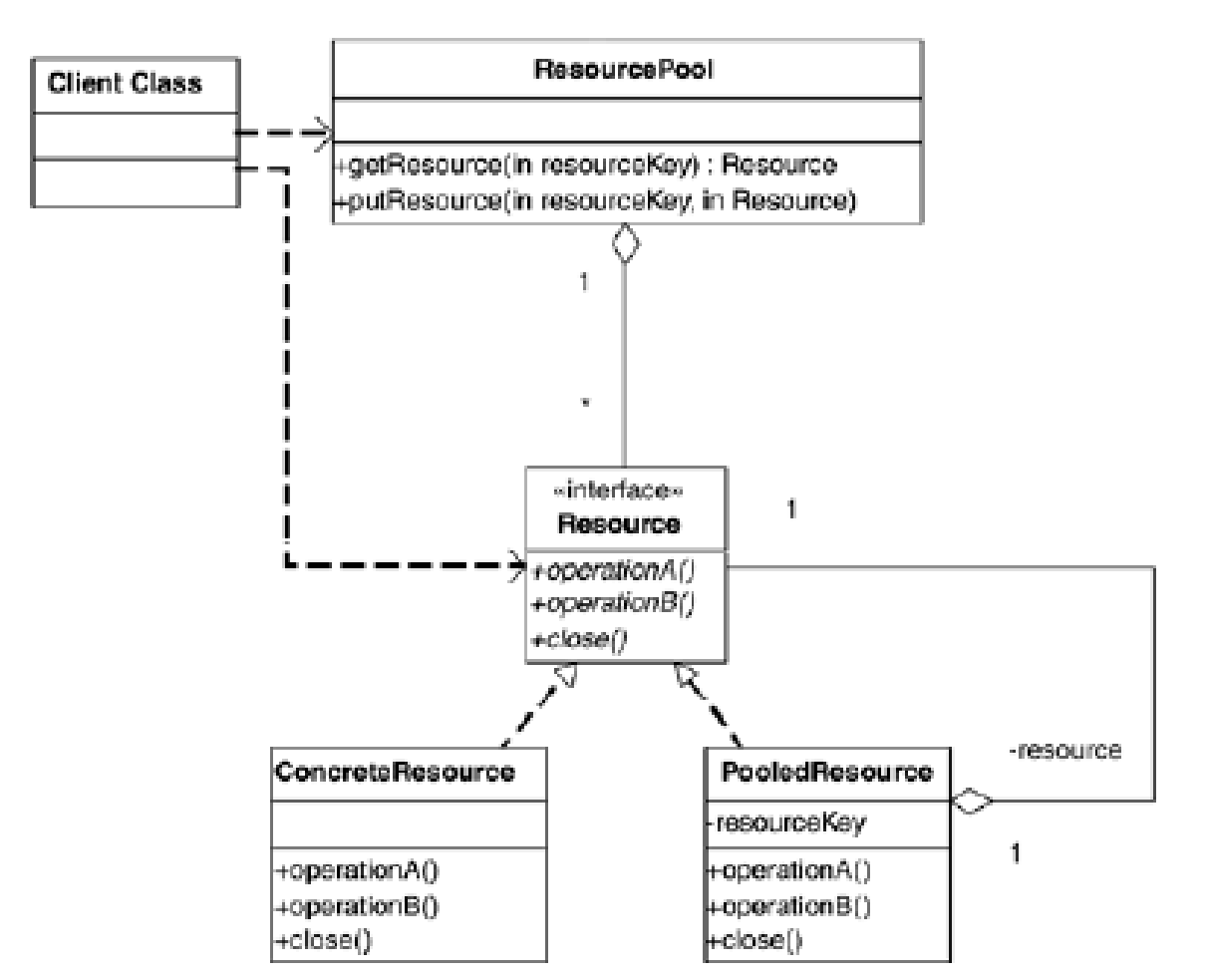
Преимущества:

- ✓ позволяет добавить функциональность ресурсу в случаях, когда невозможно изменить его код
- ✓ позволяет динамически включать и выключать дополнительную функциональность ресурса без изменения кода основного приложения
- ✓ требует минимального изменения кода основного приложения при изменении функциональности ресурса (изменяется только код инициализации ресурса)
- ✓ позволяет комбинировать несколько ресурсов и их реализации
- ✓ позволяет добавить одинаковое поведение одновременно для нескольких различных ресурсов

Недостаток:

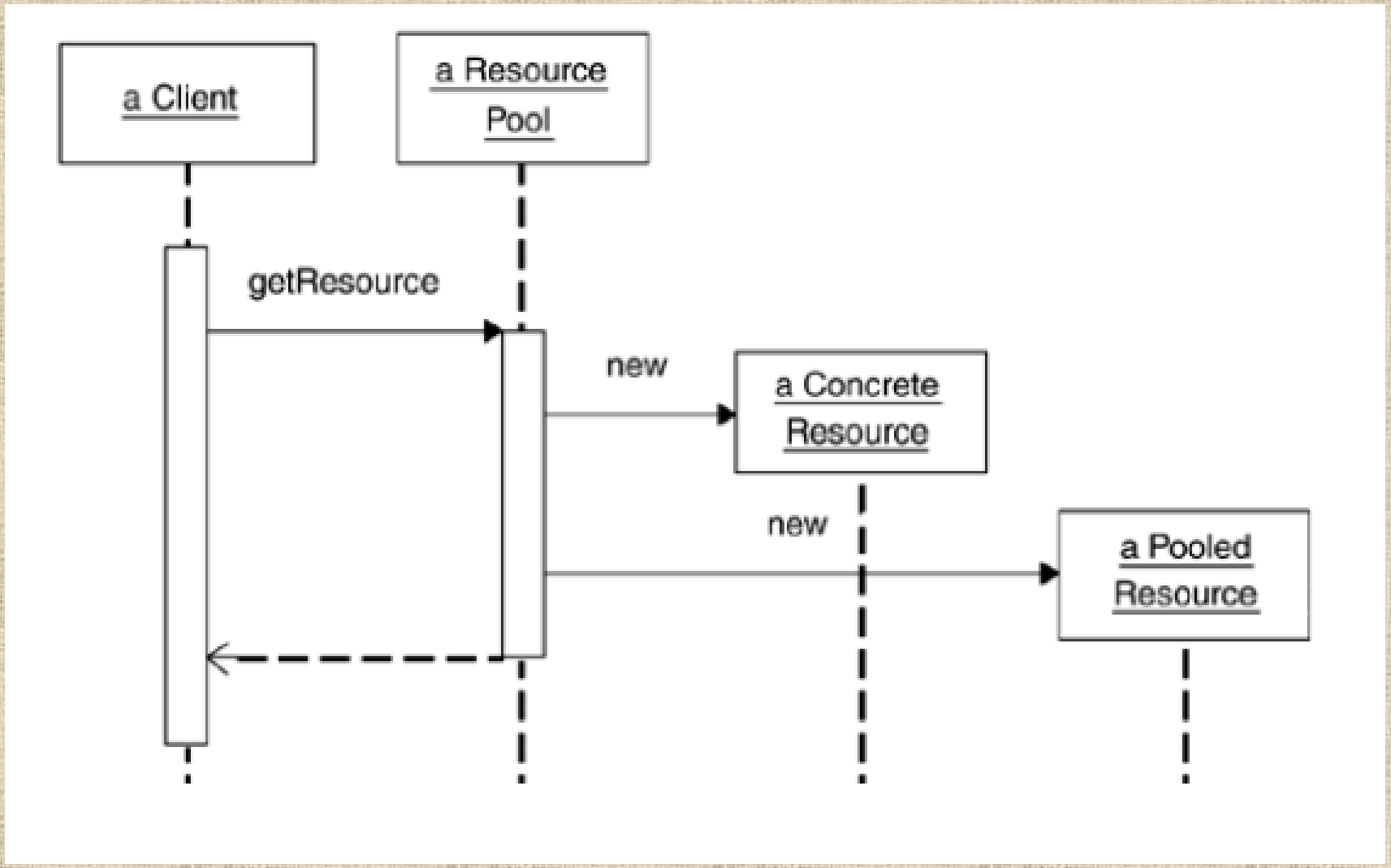
- ✓ в декораторе необходимо полностью повторить весь интерфейс конкретного ресурса

Типовое решение «Ресурсный пул» (Resource Pool)

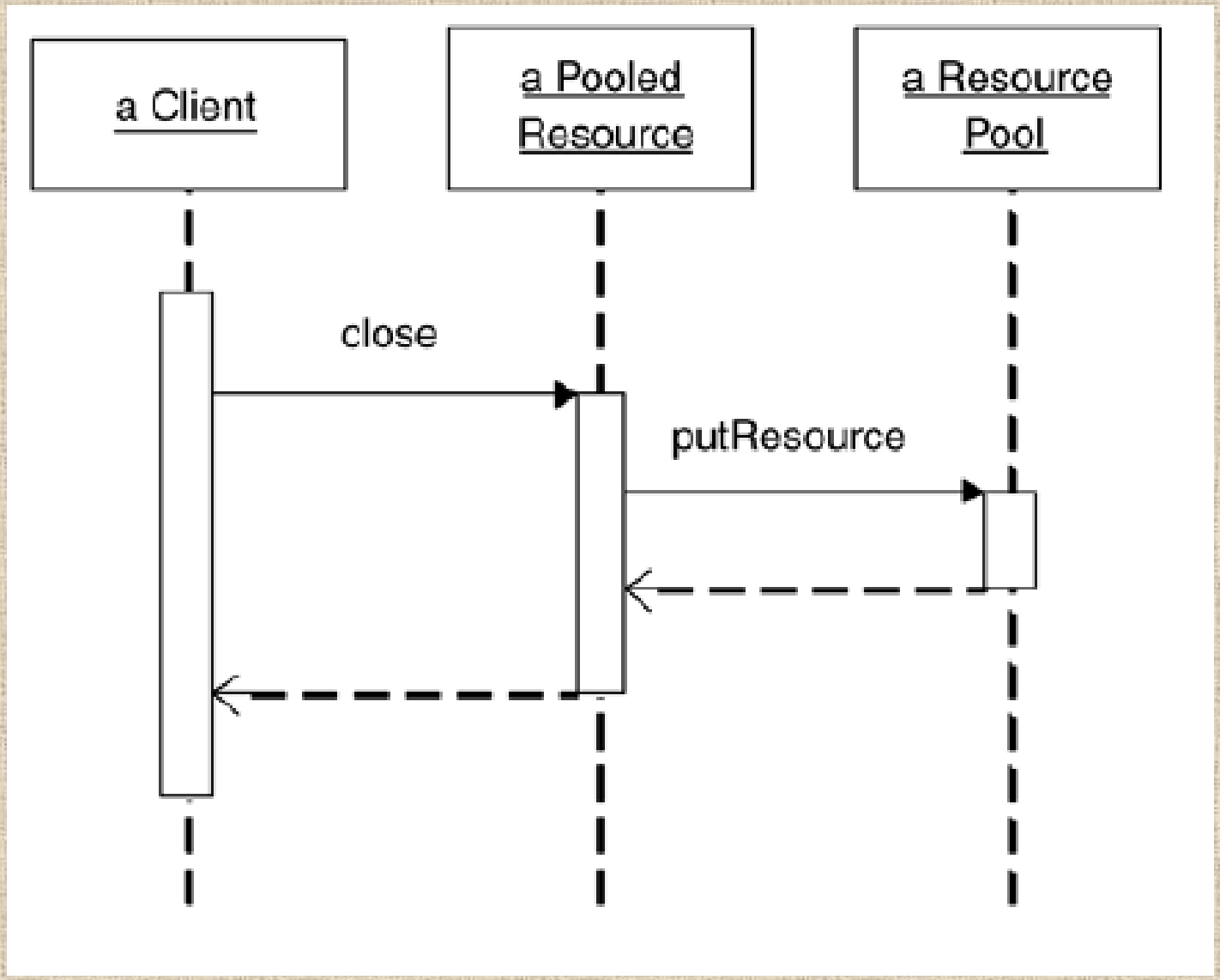


Обеспечивает повторное использование ресурсов, минимизируя накладные расходы на их инициализацию. Ресурсный пул эффективно управляет ресурсами, освобождая приложение от необходимости выполнять операции по выделению ресурсов.

Ресурсный пул: выделение нового ресурса



Ресурсный пул: освобождение ресурса



Ресурсный пул

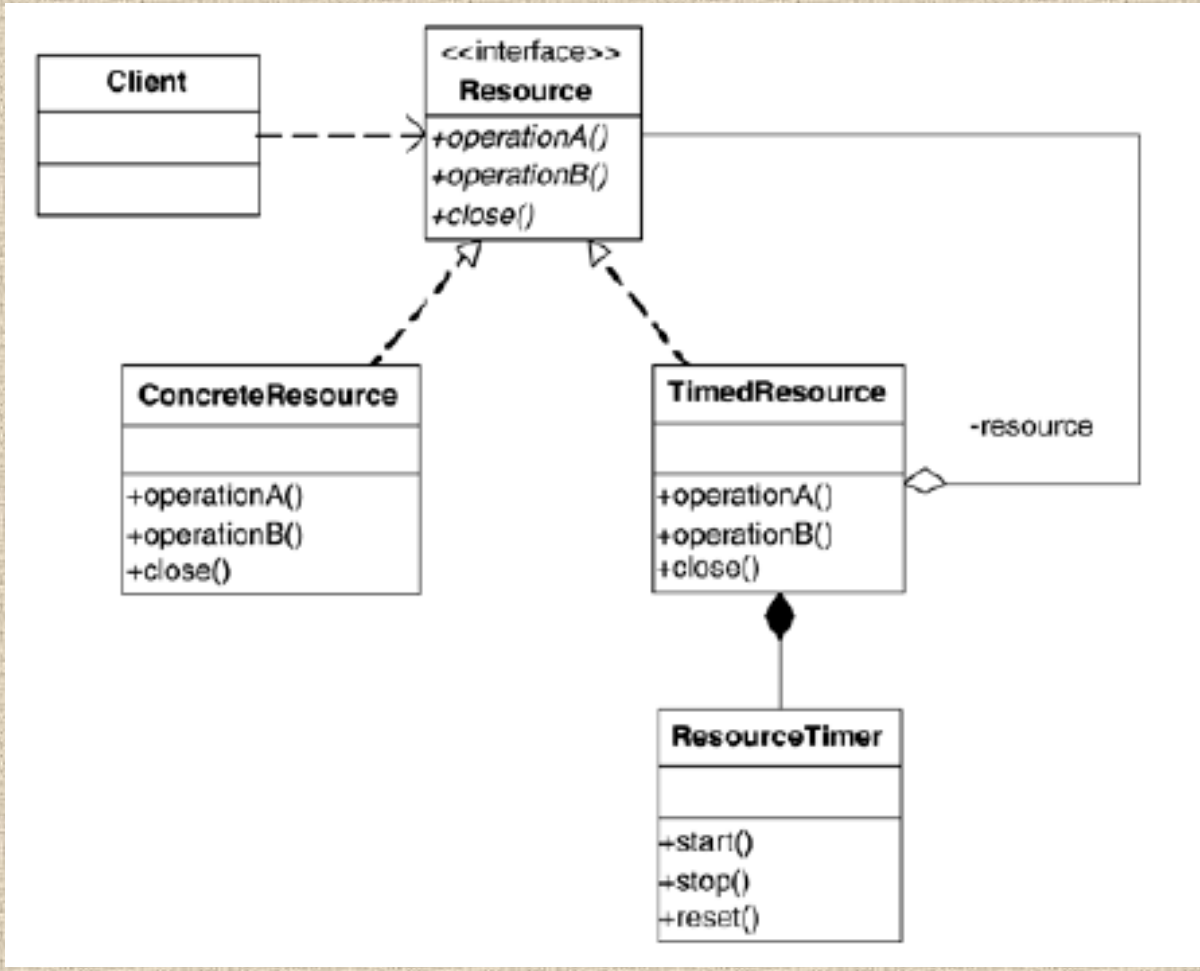
Преимущества:

- ✓ эффективное решение при работе с базами данных в многопользовательских системах
- ✓ позволяет снизить накладные расходы на инициализацию и распределение ресурсов за счет повторного использования высвобождающихся открытых ресурсов
- ✓ Централизованное управление ресурсами с возможностью оптимизации

Недостаток:

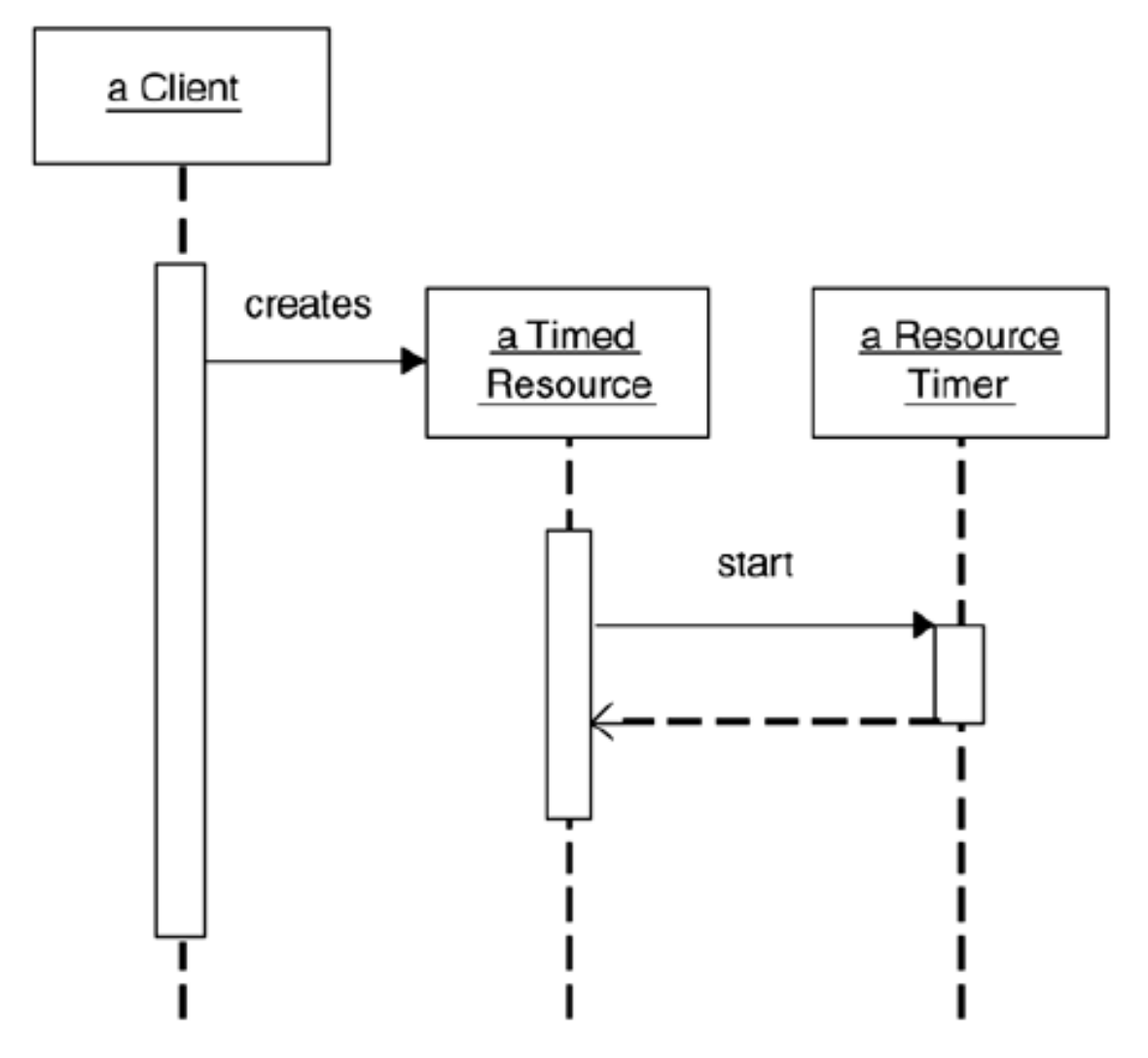
- ✓ непредсказуемое поведение пула ресурсов, например, в случаях закрытия неактивных соединений базой данной самостоятельно

Типовое решение «Ресурсный таймер» (Resource Timer)

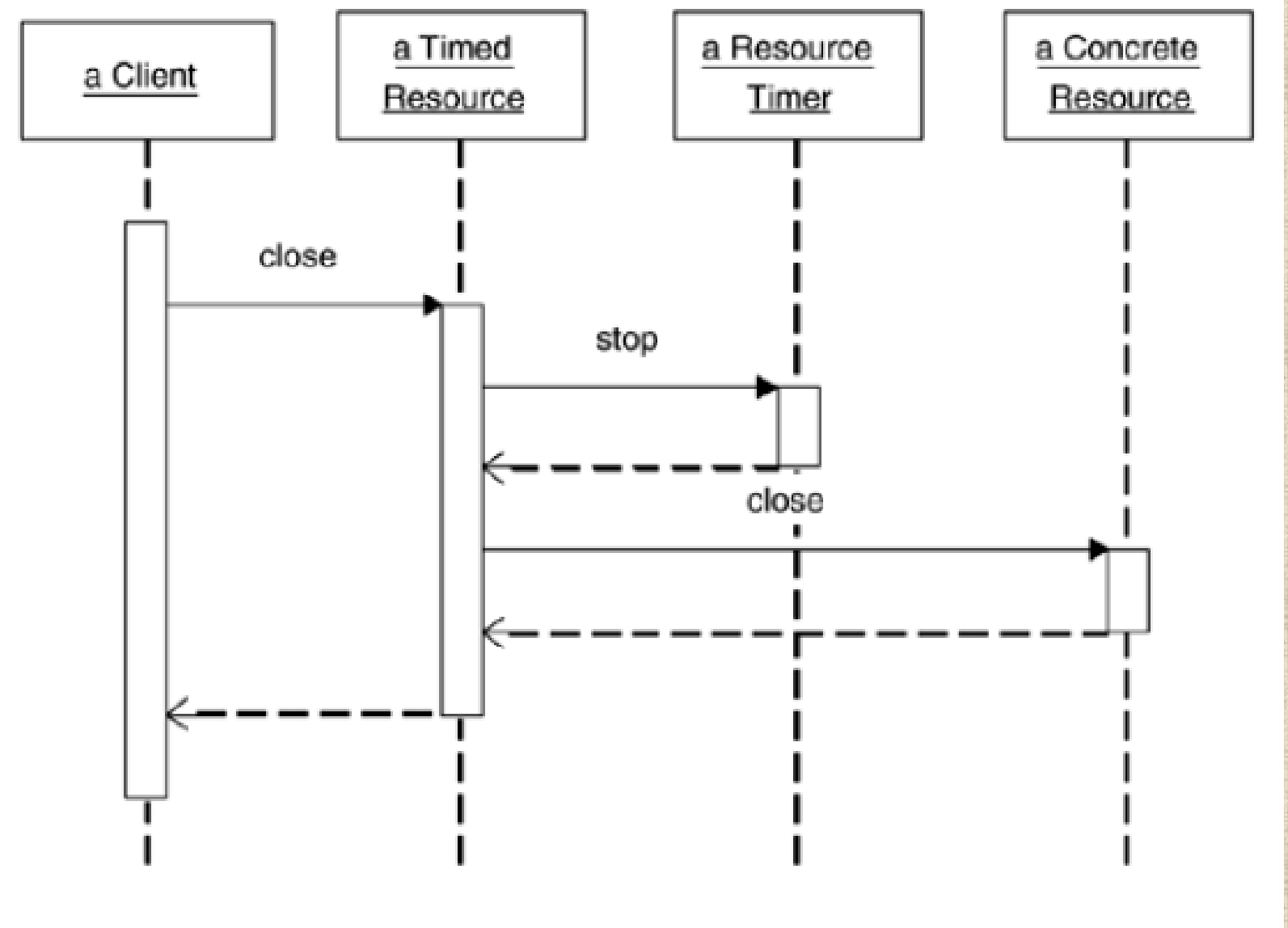


Автоматически освобождает неактивные ресурсы, решает проблему ресурсов, выделяемых на неопределенный срок.

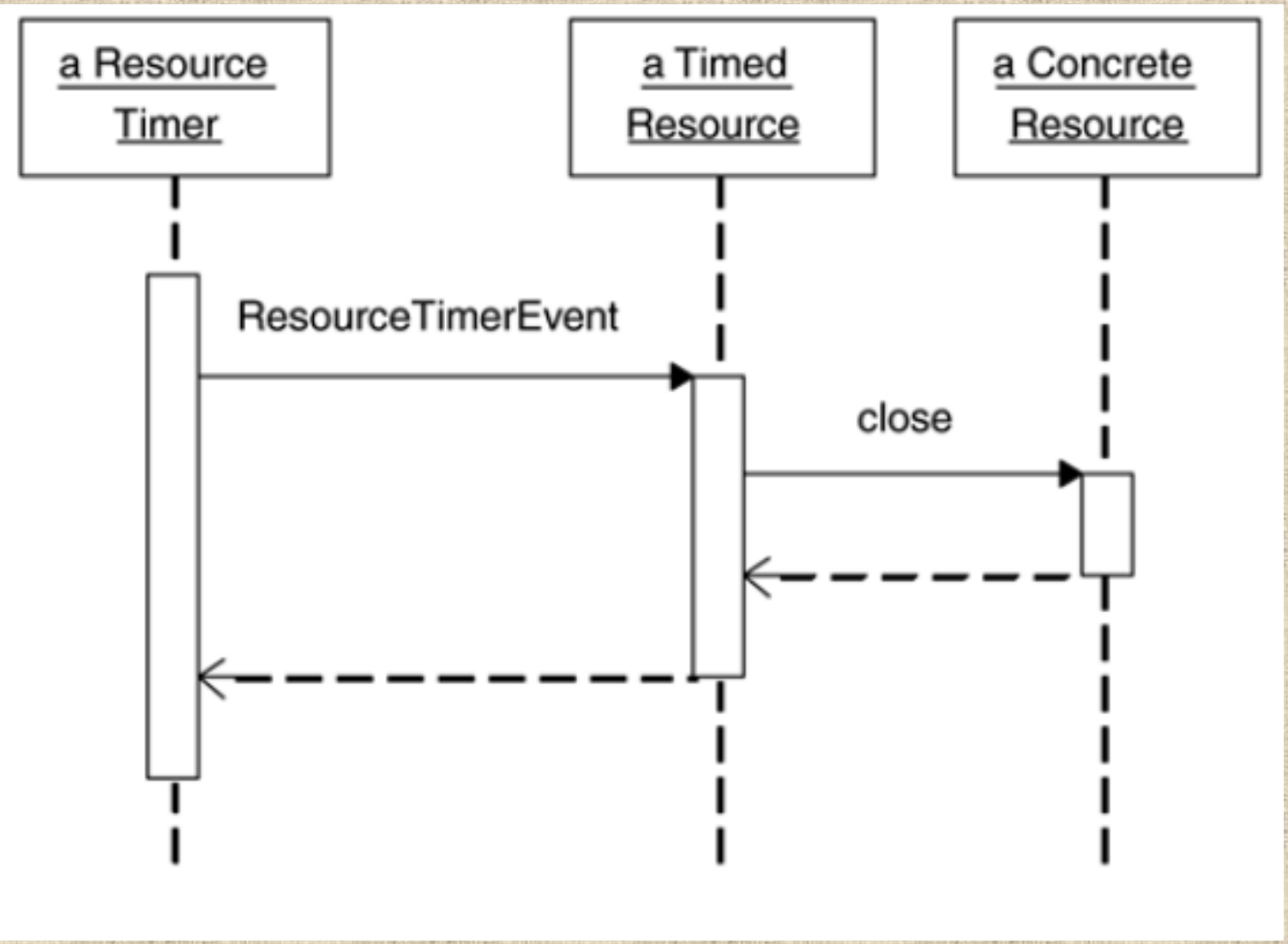
Ресурсный таймер: выделение ресурса



Ресурсный таймер: корректное закрытие ресурса



Ресурсный таймер: закрытие ресурса по времени



Ресурсный таймер

Когда применяется:

- ✓ ошибки в приложении могут привести к тому, что открытые ресурсы не будут освобождены в течении неопределенного времени
- ✓ действия пользователя могут привести к незакрытию открытого ресурса (например, при ожидании ввода пользователя)
- ✓ открытые ресурсы потребляют значительные объемы памяти

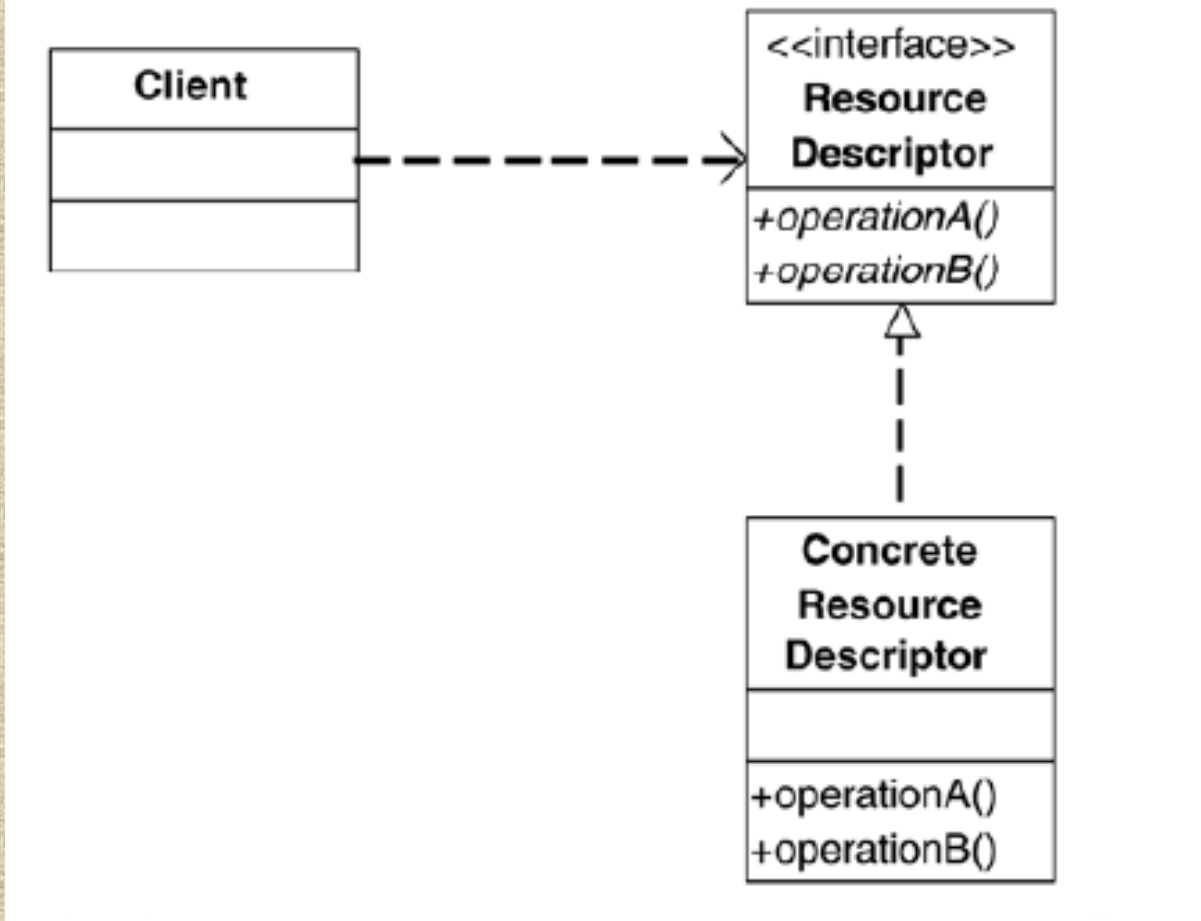
Преимущества:

- ✓ автоматически освобождает неактивные ресурсы

Недостаток:

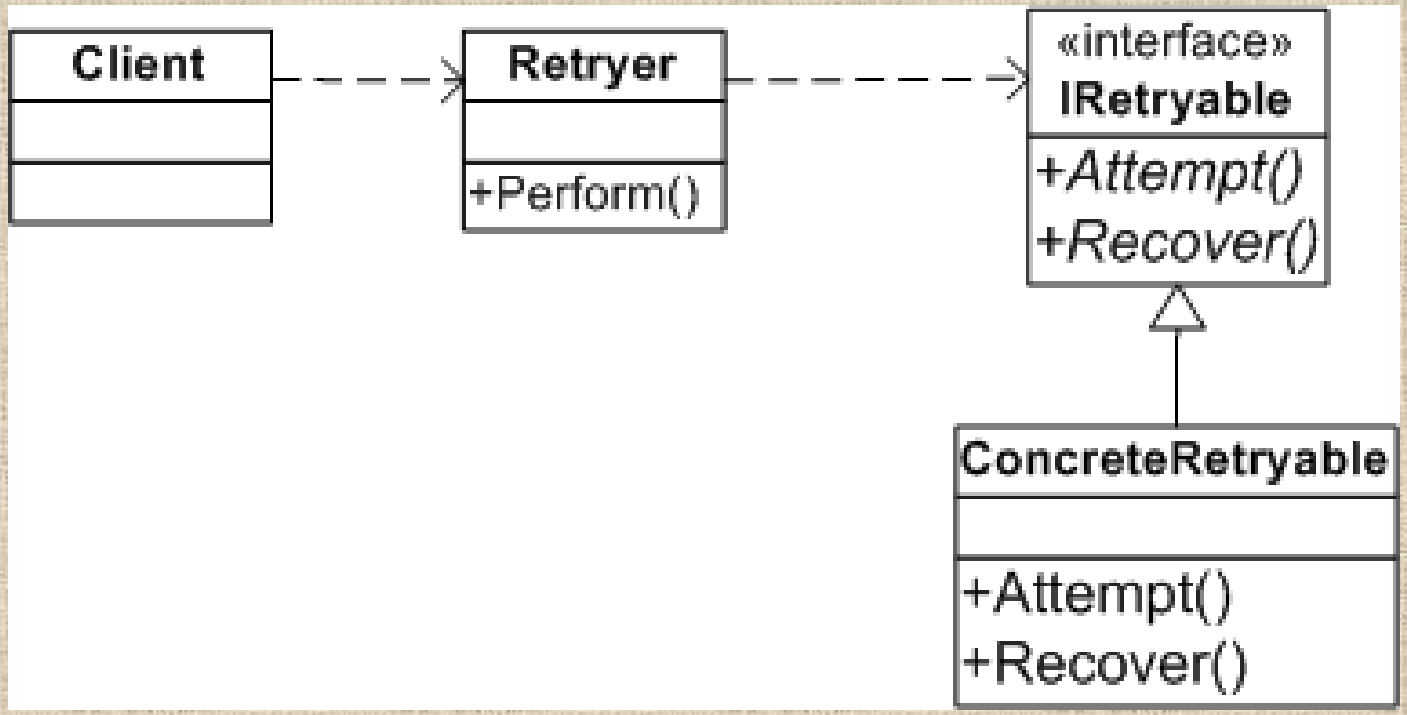
- ✓ непреднамеренные таймауты: бывают случаи, когда приложению необходимо занять ресурс и удерживать его в течение времени, превышающего пороговые значения таймера

Типовое решение «Дескриптор ресурса» (Resource Descriptor)



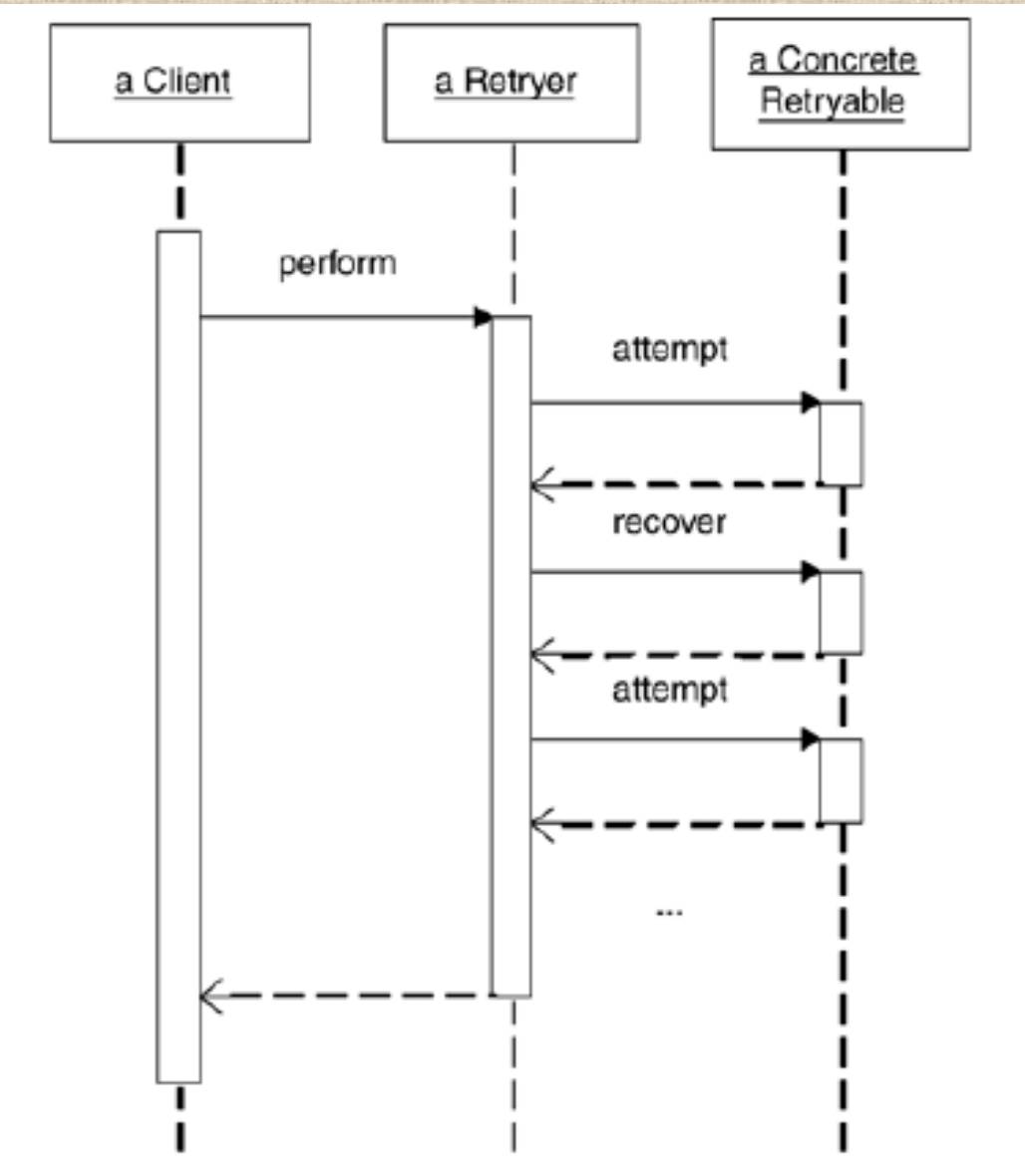
Изолирует код, зависимый от платформы и особенностей источника данных, в отдельном компоненте, предоставляя основному приложению логические операции общего вида по работе с ресурсом.

Типовое решение «Повторитель» (Retryer)



Автоматически повторяет операции, невыполнение которых ожидается при определенных условиях. Этот шаблон обеспечивает отказоустойчивость для операций доступа к данным.

Повторитель



Объектно-реляционные типовые решения

Моделирование поведения

- Единица работы (Unit of Work)
- Коллекция объектов (Identity Map)
- Загрузка по требованию (Lazy Load)

Объектно-реляционного отображения

- Отображение метаданных (Metadata Mapping)
- Объект запроса (Query Object)
- Хранилище (Repository)

Моделирование структуры

- Поле идентификации (Identity Field)
- Отображение внешних ключей (Foreign Key Mapping)
- Отображение с помощью таблицы ассоциаций (Association Table Mapping)
- Отображение зависимых объектов (Dependent Mapping)
- Внедренное значение (Embedded Value)
- Наследование с одной таблицей (Single Table Inheritance)
- Наследование с таблицами для каждого класса (Class Table Inheritance)
- Наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance)

Проблемы взаимного отображения объектов и реляционных структур

1. Обеспечение однозначного соответствия между данными в таблицах базы данных и свойствами (атрибутами) объектов.

2.Связи объектов и связи таблиц базы данных реализуются по-разному

a) Объекты манипулируют связями, сохраняя ссылки в виде адресов памяти. В реляционных базах данных связь одной таблицы с другой задается путем формирования соответствующего внешнего ключа (foreign key)

b) С помощью структуры коллекции объект способен сохранить множество ссылок из одного поля на другие, тогда как правила нормализации таблиц базы данных допускают применение только однозначных ссылок

c) Связь между объектами категории "многие ко многим", т.е. коллекции ссылок присутствуют "с обеих сторон" связи. В реляционных базах данных проблема описания такой структуры преодолевается за счет создания дополнительной таблицы, содержащей пары ключей связанных сущностей.

d) Представление отношения наследования между классами (объектами).

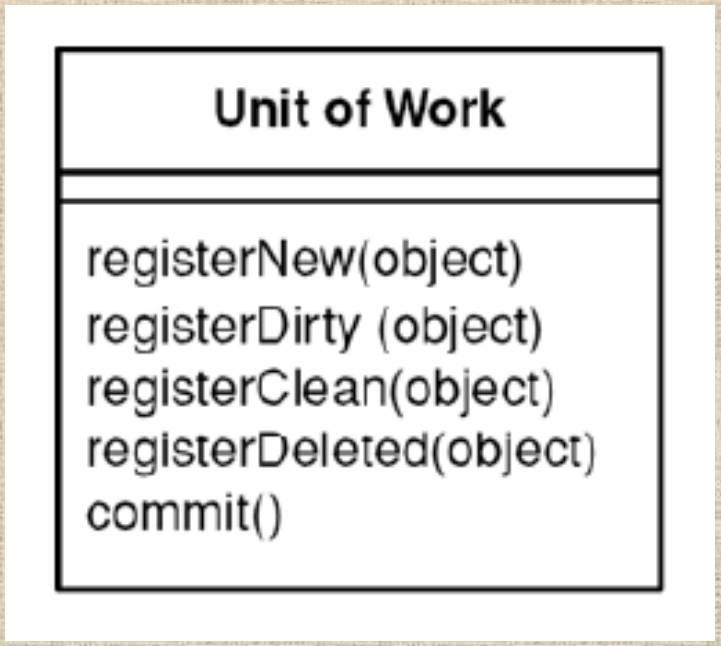
3. Проблемы параллельного изменения объектов, изменения базы данных и объектов

Пример:

В объекте, представляющем сущность "заказ", естественно предусмотреть коллекцию ссылок на объекты, описывающие заказываемые товары, причем последним нет необходимости ссылаться на "родительский" объект заказа.

В схеме базы данных запись в таблице товаров должна содержать внешний ключ, указывающий на запись в таблице заказов, поскольку заказ не может иметь многозначного поля.

Типовое решение «Единица работы» (Work of Unit)



Позволяет контролировать все действия, выполняемые в рамках бизнес-транзакции, которые так или иначе связаны с базой данных. По завершении всех действий определяет окончательные результаты работы, которые и будут внесены в базу данных.

Изменения в базу данных можно вносить при каждом изменении содержимого объектной модели, однако это неизбежно выльется в гигантское количество мелких обращений к базе данных, что значительно снизит производительность.

Единица работы

Как это работает

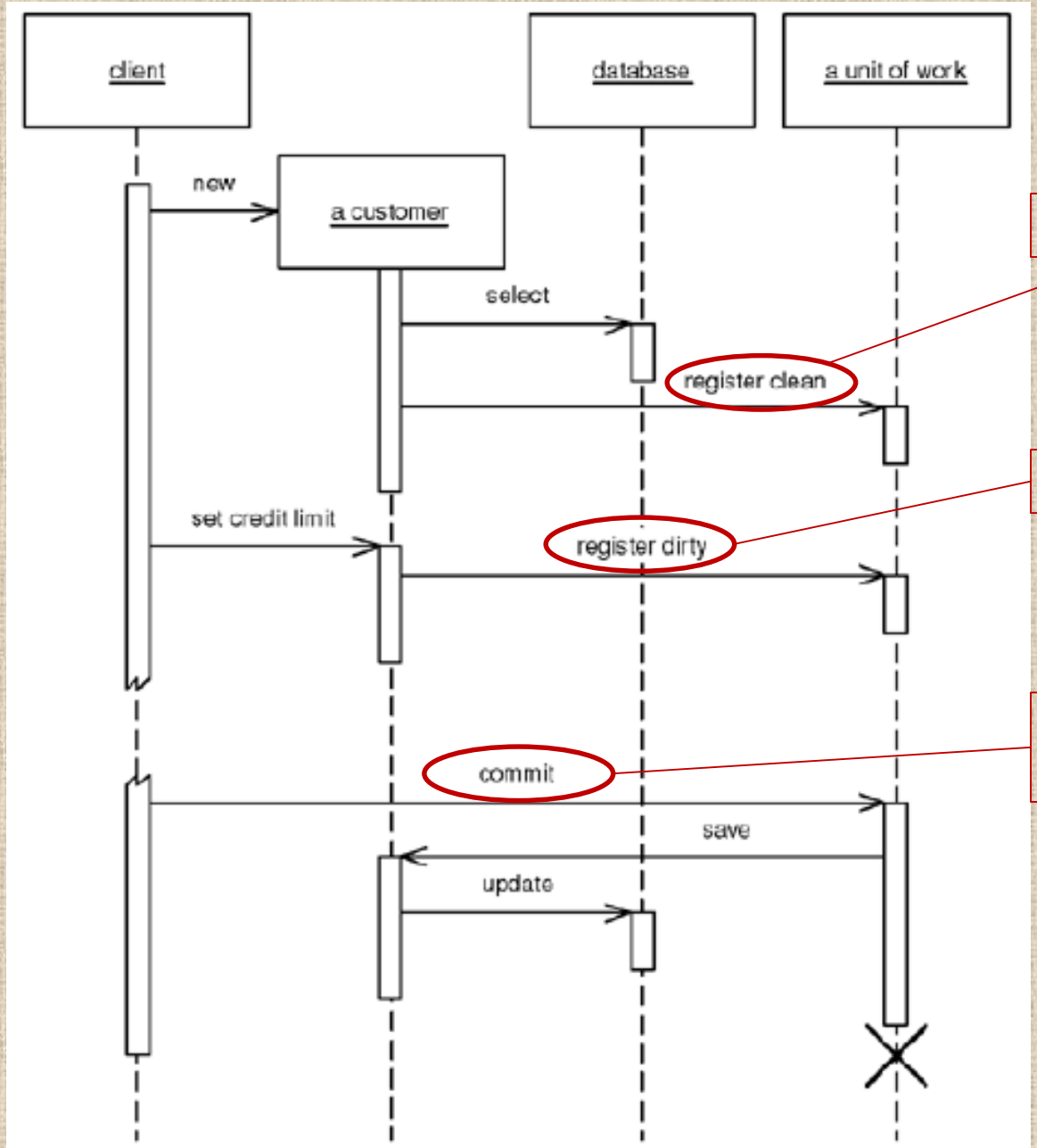
Как только делается что-нибудь, что может затронуть содержимое базы данных, создается **единица работы**, которая должна контролировать все выполняемые изменения.

Каждый раз, создавая, изменяя или удаляя объект, генерируется сообщение об этом единице работы. Кроме того, следует сообщать, какие объекты были считаны из базы данных, чтобы не допустить их несогласованности (для чего единица работы проверяет, не были ли запрошенные объекты изменены во время считывания).

Когда необходимо зафиксировать сделанные изменения, единица работы определяет, что ей нужно сделать. Она сама открывает транзакцию, выполняет всю необходимую проверку на наличие параллельных операций и записывает изменения в базу данных. Разработчики приложений никогда явно не вызывают методы, выполняющие обновления базы данных. Таким образом, им не приходится отслеживать, что было изменено, или беспокоиться о том, в каком порядке необходимо выполнить нужные действия, чтобы не нарушить целостность на уровне ссылок, — единица работы сделает это за них.

Чтобы единица работы действительно вела себя подобным образом, ей должно быть известно, за какими объектами необходимо следить. Об этом ей может сообщить оператор, выполняющий изменение объекта, или же сам объект

Единица работы: регистрация посредствам изменяемого объекта

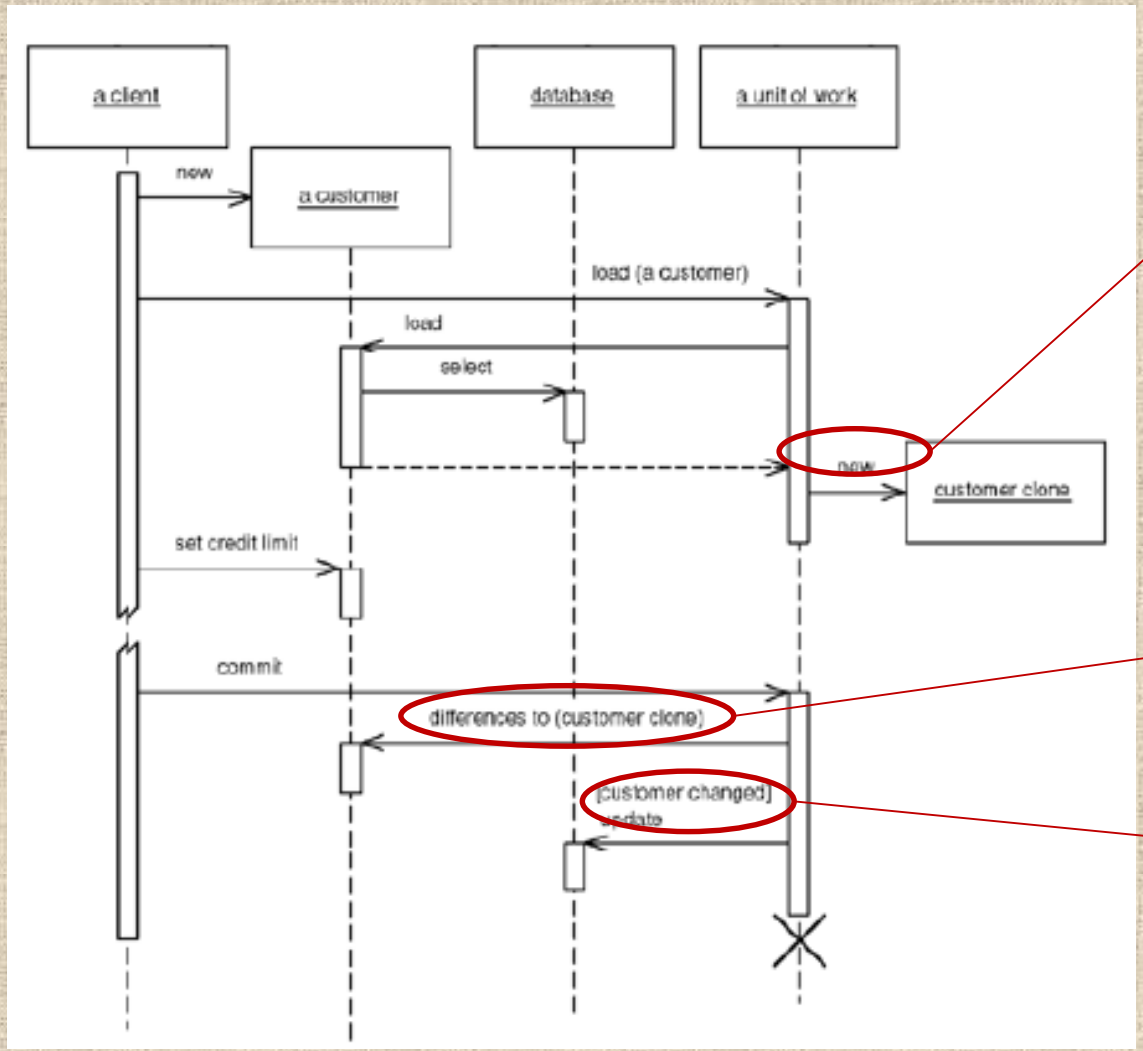


Регистрация исходного объекта

Регистрация измененного объекта

Сохранение изменений в базе данных

Единица работы: контроллер доступа к базе данных



Создание копии реального объекта

Проверка наличия изменений объекта

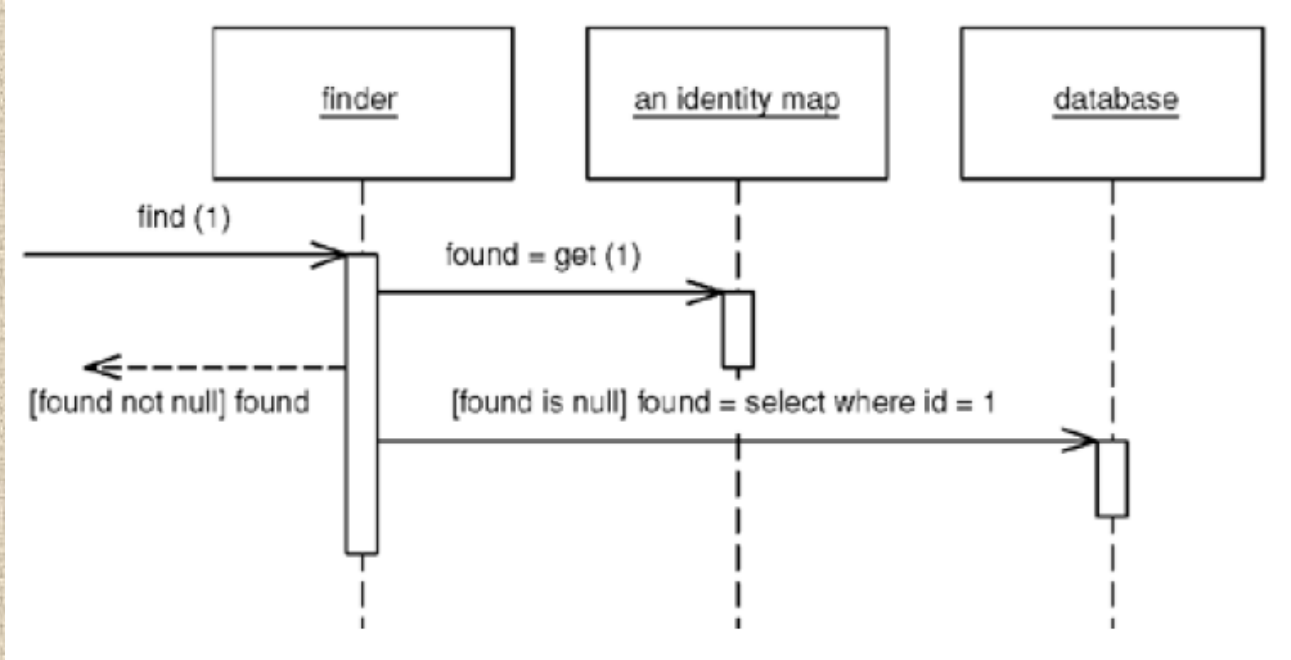
Сохранение только изменений объекта

Единица работы

Преимущества:

- ✓ позволяет сократить количество обращений к базе данных
- ✓ позволяет эффективно реализовать процедуры отката изменений объектов
- ✓ позволяет реализовать процедуру контроля порядка обновления таблиц базы данных с учетом контроля целостности и взаимозависимостей объектов (таблиц)
- ✓ эффективная реализация пакетных обновлений – отправка нескольких sql-команд как единого целого
- ✓ может применяться не только с базами данных, но и с любыми ресурсами транзакций, для манипулирования очередями сообщений и мониторами транзакций

Типовое решение «Коллекция объектов» (Identity Map)



Гарантирует, что каждый объект будет загружен из базы данных только один раз, сохраняя загруженный объект в специальной коллекции. При получении запроса просматривает коллекцию в поисках нужного объекта.

Позволяет избежать проблем связанных с повторной одновременной загрузкой одних и тех же данных в разные объекты.

Коллекция объектов

Как это работает

В случае простой изоморфной схемы для каждой таблицы базы данных создается своя коллекция. Прежде чем загружать объект из базы данных, необходимо проверить, нет ли его в коллекции объектов?

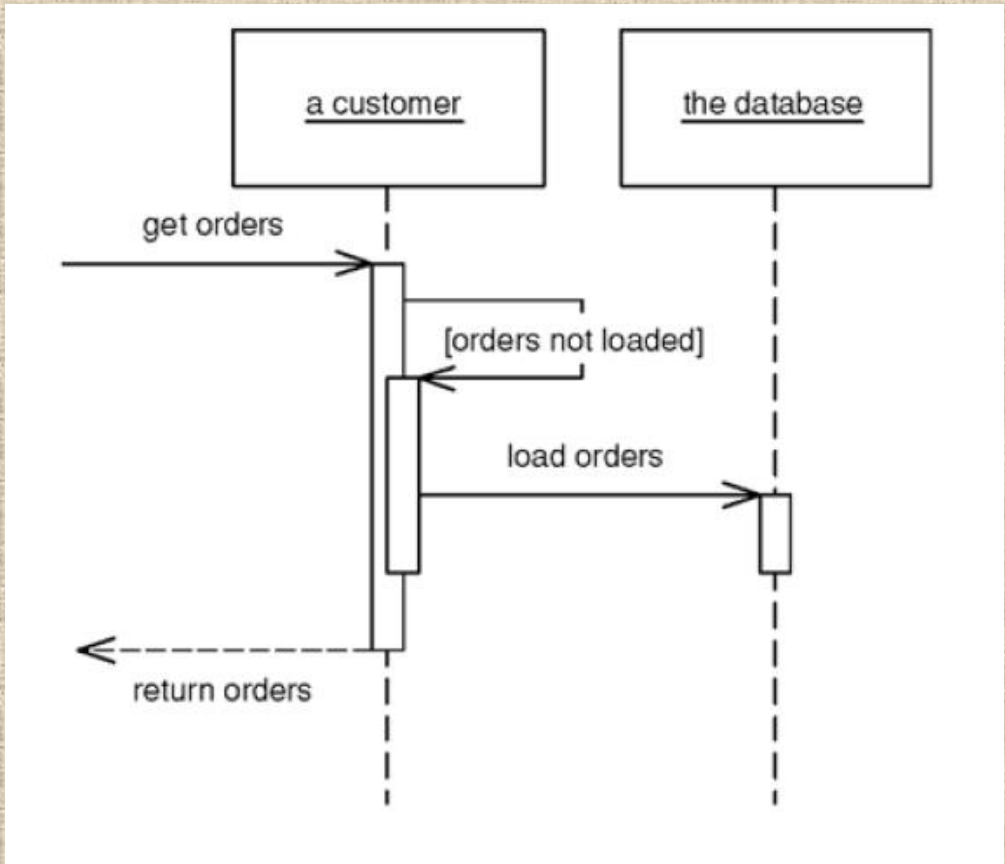
Если в ней обнаружится объект, в точности соответствующий тому, который нужен, то возвращается найденный объект.

Если же подходящего объекта не оказалось, то объект загружается из базы данных и помещается в коллекцию для дальнейшего использования.

Проблемы:

1. Выбор ключей для поиска объектов в коллекции
2. Явная или универсальная коллекция. В явной коллекции используются явные (специализированные) методы поиска. В универсальной коллекции используются общие методы поиска, которым дополнительно к идентификатору объекта передается тип объекта. Универсальная коллекция применима только для объектов, имеющих ключ одного типа.
3. Сколько создавать коллекций: по одной для каждого класса или одну на сеанс? Одна коллекция на сеанс может быть создана только, если все таблицы базы данных имеют ключи одного типа. Большое количество коллекций создает проблемы организации наследования объектов.
4. Куда поместить коллекцию объектов? В объект, специфичный для сеанса.

Типовое решение «Загрузка по требованию» (Lazy Load)



Объект, который не содержит все требующиеся данные, однако может загрузить их в случае необходимости.

Загрузку данных в оперативную память следует организовать таким образом, чтобы при загрузке интересующего объекта из базы данных автоматически извлекались и другие связанные с ним объекты. При этом может быть выгружена вся база данных!

Типовое решение «загрузка по требованию» прерывает процесс загрузки, оставляя соответствующую метку в структуре объектов. Это позволяет загрузить необходимые данные только тогда, когда они действительно понадобятся.

Загрузка по требованию

Как это работает (варианты реализации)

1. **Инициализация по требованию.** При каждой попытке доступа к полю выполняется проверка, не содержит ли оно значение NULL. ЕСЛИ поле содержит NULL, метод доступа загружает значение поля и лишь затем его возвращает. Это может быть реализовано только в том случае, если поле является самоинкапсулированным, т.е. если доступ к такому полю осуществляется только посредством get-метода (даже в пределах самого класса).

2. **Виртуальный прокси-объект.** Виртуальный прокси-объект имитирует объект, являющийся значением поля, однако в действительности ничего в себе не содержит. В этом случае загрузка реального объекта будет выполнена только тогда, когда будет вызван один из методов виртуального прокси-объекта. Преимуществом виртуального прокси-объекта является то, что он "выглядит" точь-в-точь как реальный объект, который должен находиться в данном поле. Тем не менее это не настоящий объект, поэтому возникает проблема идентификации.

3. **Фиктивный объект.** Фиктивный объект— это реальный объект с неполным состоянием. Когда подобный объект загружается из базы данных, он содержит только свой идентификатор. При первой же попытке доступа к одному из его полей объект загружает значения всех остальных полей.

Загрузка по требованию

Проблемы:

1. **Реализация наследования.** При использовании прокси-объектов и фиктивных объектов необходимо заранее знать тип создаваемого объекта. А это не всегда возможно на этапе разработки и компиляции.

2. **Волнообразная загрузка.** Например, при заполнении коллекций. Заполнив коллекцию неполными объектами, просмотр каждого объекта приведет к обращению к базе данных.

3. Бессмысленно применять загрузку по требованию, чтобы извлечь значения полей из одной и той же строки таблицы базы данных.

4. Загрузку по требованию необходимо применять в тех случаях, когда интерфейс пользователя требует отдельного обращения к базе данных для извлечения информации, которая не используется с основным объектом.

5. Применение загрузки по требованию существенно усложняет приложение.

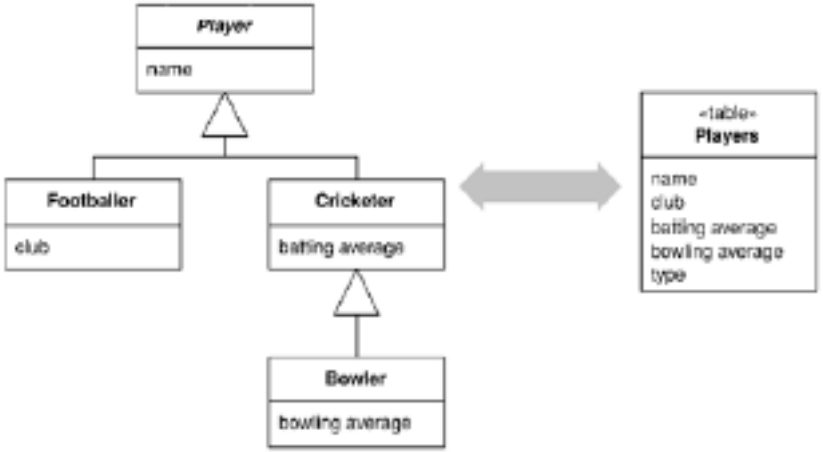
Объектно-реляционные типовые решения для моделирования структуры

- Поле идентификации (Identity Field)
- Отображение внешних ключей (Foreign Key Mapping)
- Отображение с помощью таблицы ассоциаций (Association Table Mapping)
- Отображение зависимых объектов (Dependent Mapping)
- Внедренное значение (Embedded Value)
- Наследование с одной таблицей (Single Table Inheritance)
- Наследование с таблицами для каждого класса (Class Table Inheritance)
- Наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance)

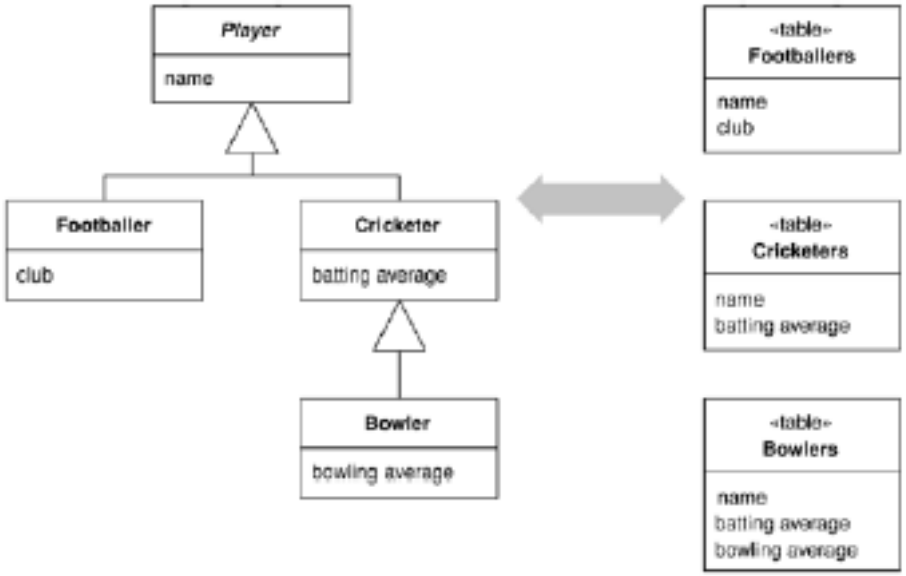
Проблема отображения отношений

Проблема отображения наследования

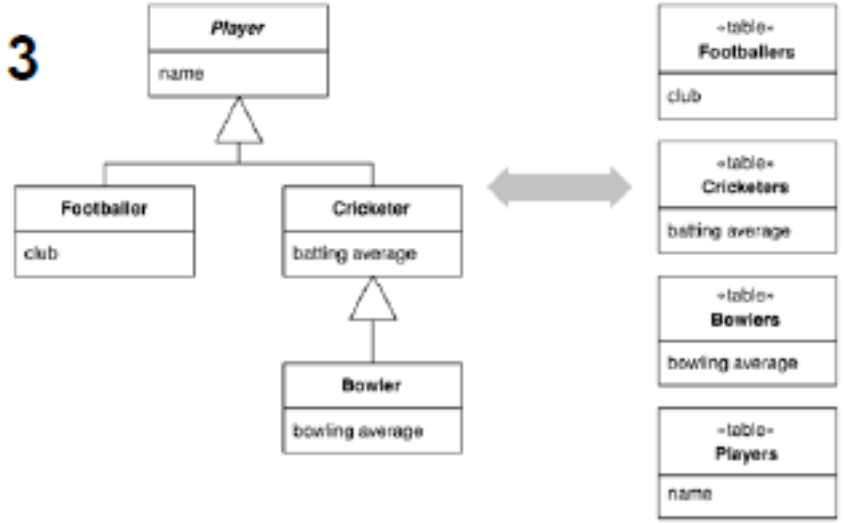
1



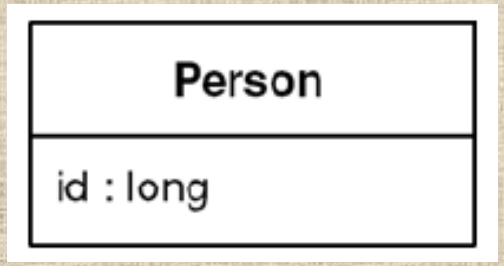
2



3



Типовое решение «Поле идентификации» (Identity Field)



Сохраняет идентификатор записи базы данных в поле объекта для поддержки соответствия между объектом приложения и строкой базы данных.

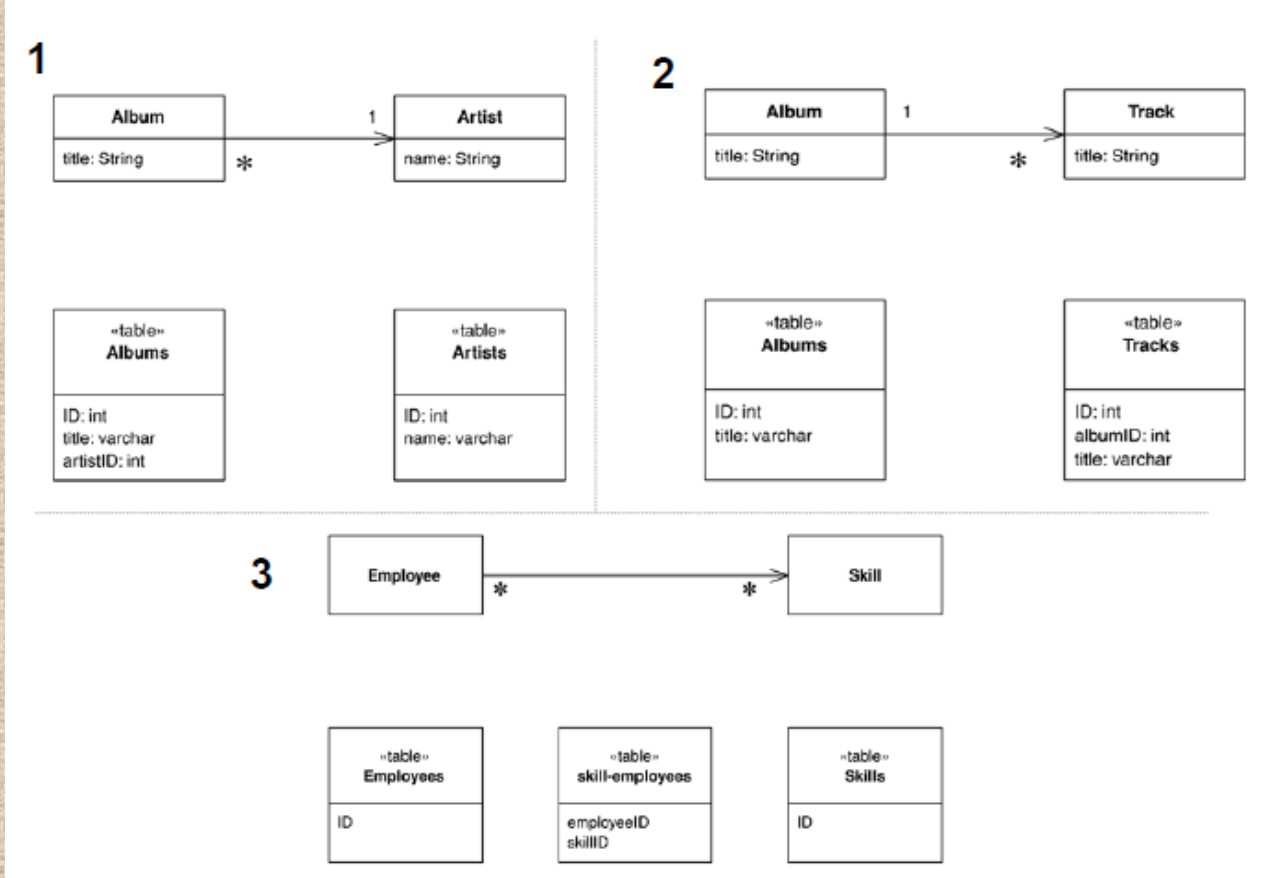
Поле идентификации: сложные вопросы

1. **Выбор ключа.**
   a) Использовать значащий ключ (значение предметной области) или незначащий ключ (случайное уникальное число) ? Рекомендуется использовать незначащие ключи.
   b) Использовать простые или составные ключи? Составной ключ целесообразно использовать, когда одна таблица имеет смысл в контексте другой таблицы. Простые ключи более универсальные.
   c) Какой тип ключа использовать? Рекомендация – выбирать такой тип, для которого операция сравнения выполняется максимально быстро.
   d) Использовать ключи уникальные в пределах таблиц или в пределах базы данных? Ключи, уникальные в пределах базы данных позволяют использовать общую коллекцию объектов.

2. **Представление поля идентификации в объекте**.
   a) Использовать поле, тип которого соответствует типу ключа базы данных.
   b) При использовании в базе данных составных ключей, необходимо создавать классы ключей для их хранения в объектах.
   c) Можно использовать универсальный класс для представления всех ключей.

3. **Вычисление нового значения ключа.**
   a) Генерация значения ключа с помощью средств базы данных. Недостаток – значение ключа новой записи невозможно узнать в рамках одной транзакции
   b) Использовать счетчик в базе данных
   c) Глобальный уникальный идентификатор (GUID), сгенерированный на компьютере.
   d) Использовать таблицу ключей в базе данных, с отдельной транзакцией

Типовое решение «Отображение внешних ключей» (Foreign Key Mapping)



Отображает ассоциации между объектами (реализованные объектными ссылками) на ссылки внешнего ключа между таблицами базы данных.

Основные сложности связаны с ссылками на коллекции объектов при выполнении операций обновления (вставки, удаления). Как определить, какие изменения внести в базу данных после внесения изменений в коллекции объектов?

Отображение внешних ключей: сложные вопросы

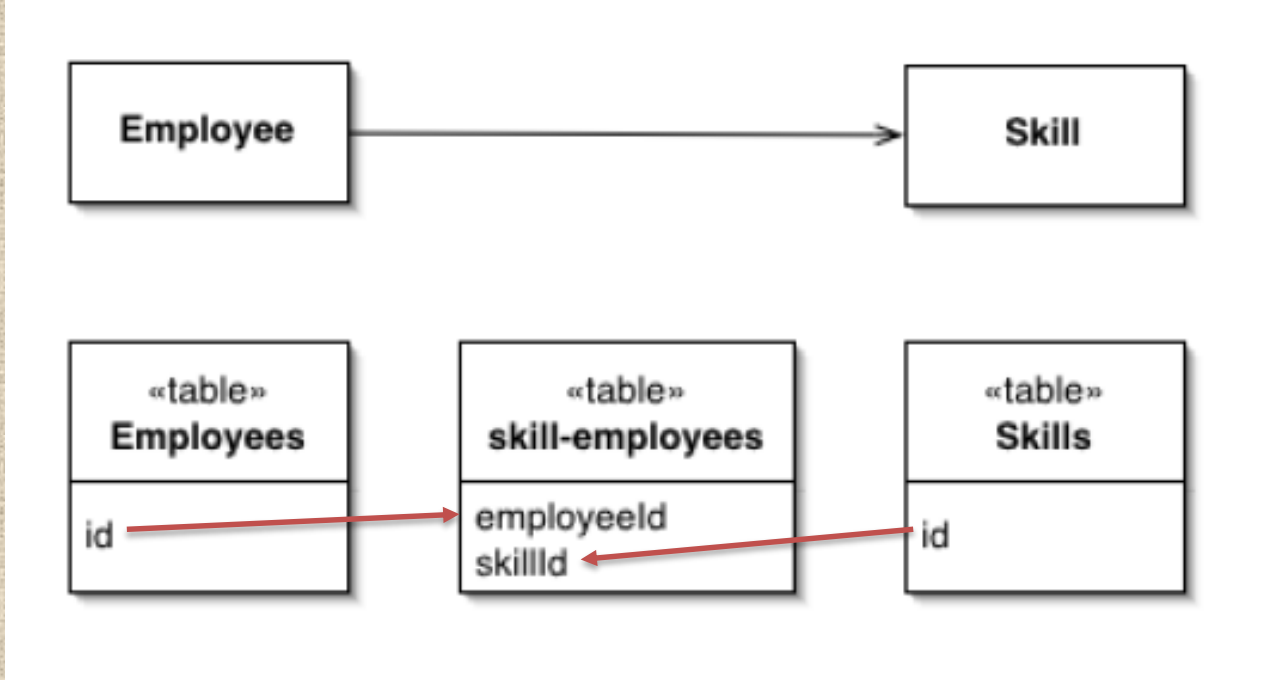
1. **1:1.** Воспользоваться полем идентификации. Каждому объекту соответствует определенное значение ключа таблицы базы данных. Если два объекта связаны ассоциацией, в базе данных ее можно заменить внешним ключом.

2. **1:n**.
   a) *Удаление и вставка*. Удалить из базы данных все объекты, ссылающиеся на общий объект, а затем вставить все те дочерние объекты, которые на данный момент связаны с общим объектом (в памяти).
   b) *Добавление обратного указателя*. Создание в дочернем объекте ссылки на общий объект (связь становится двунаправленной).
   c) Сравнение коллекций – текущее состояние сравнивается с состоянием в базе данных.

3. **Циклические ссылки (пример: заказ – покупатель – платежи – заказы).**
   a) Загрузка по требованию (Lazy Load)
   b) Использовать коллекцию объектов (Identity Map)

Ограничение: с помощью отображения внешних ключей нельзя моделировать связи «многие ко многим». Внешние ключи являются одномерными значениями, а из определения первой нормальной формы следует, что в одном поле нельзя хранить множественные значения внешних ключей.

Типовое решение «Отображение с помощью таблицы ассоциаций» (Association Table Mapping)



Сохраняет множество ассоциаций в виде таблицы, содержащей внешние ключи таблиц, связанных ассоциациями.

Решение : создать дополнительную таблицу отношений, а затем воспользоваться типовым решением «отображение с помощью таблицы ассоциаций», чтобы отобразить многозначное поле на таблицу отношений.

Отображение с помощью таблицы ассоциаций

Как это работает:

Основная идея - хранение ассоциаций в таблице отношений. Последняя содержит только значения внешних ключей двух таблиц, связанных отношением. Каждой паре взаимосвязанных объектов соответствует одна строка таблицы отношений.

Таблице отношений не соответствует объект приложения, вследствие чего у нее нет идентификатора объекта. Первичным ключом данной таблицы является совокупность двух первичных ключей таблиц, которые связаны отношениями.

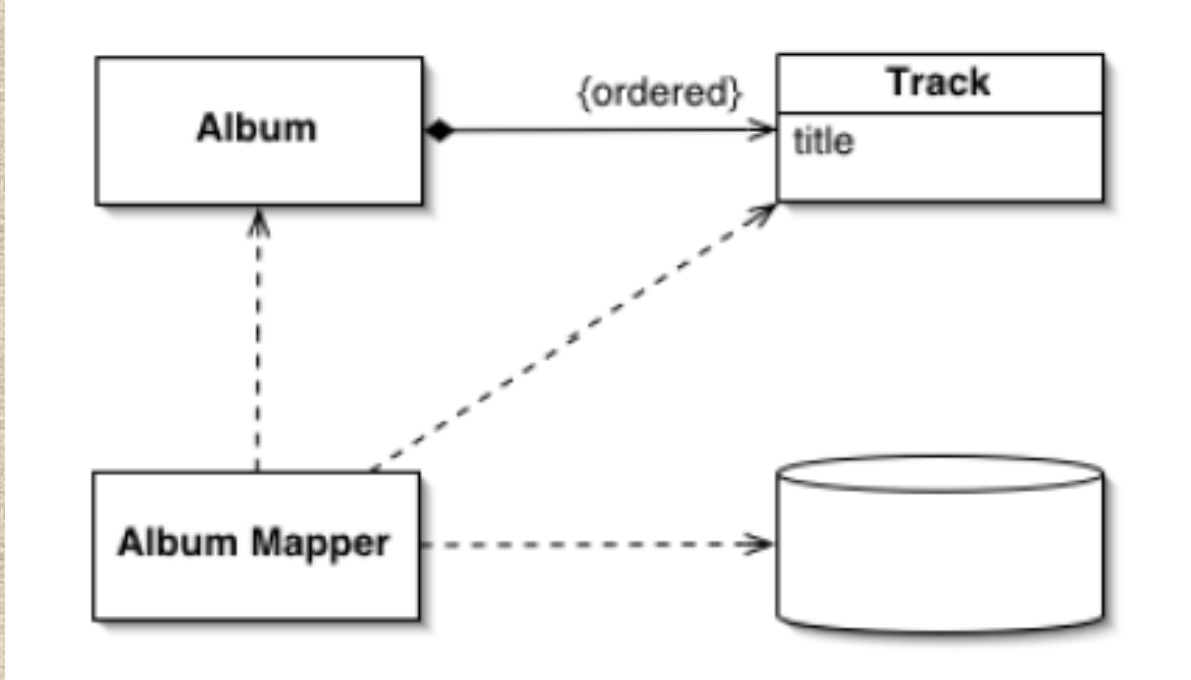
Чтобы загрузить данные из таблицы отношений, необходимо выполнить два запроса:

1. Из таблицы отношений выбираются идентификаторы объектов, связанных с заданным
2. Из таблицы сущности выбираются все строки, соответствующие найденным на предыдущем шаге идентификаторам.

Данное типовое решение незаменимо при отсутствии полного контроля над схемой базы данных и возможности ее изменения: необходимо связать две таблицы, не добавляя столбцов в ним.

В таблицу отношений можно добавлять поля с характеристиками отношения.

Типовое решение «Отображение зависимых объектов» (Dependent Mapping)



Передает некоторому классу полномочия по выполнению отображения для дочернего класса.

Некоторые объекты в силу своей семантики применяются в контексте других объектов. Например, композиции альбома могут загружаться или сохраняться тогда же, когда и сам альбом. Если на композиции альбома не ссылается никакая другая таблица базы данных, процедуру отображения можно значительно упростить, передав полномочия по выполнению отображения для композиций объекту, выполняющему отображение для альбома.

Отображение зависимых объектов

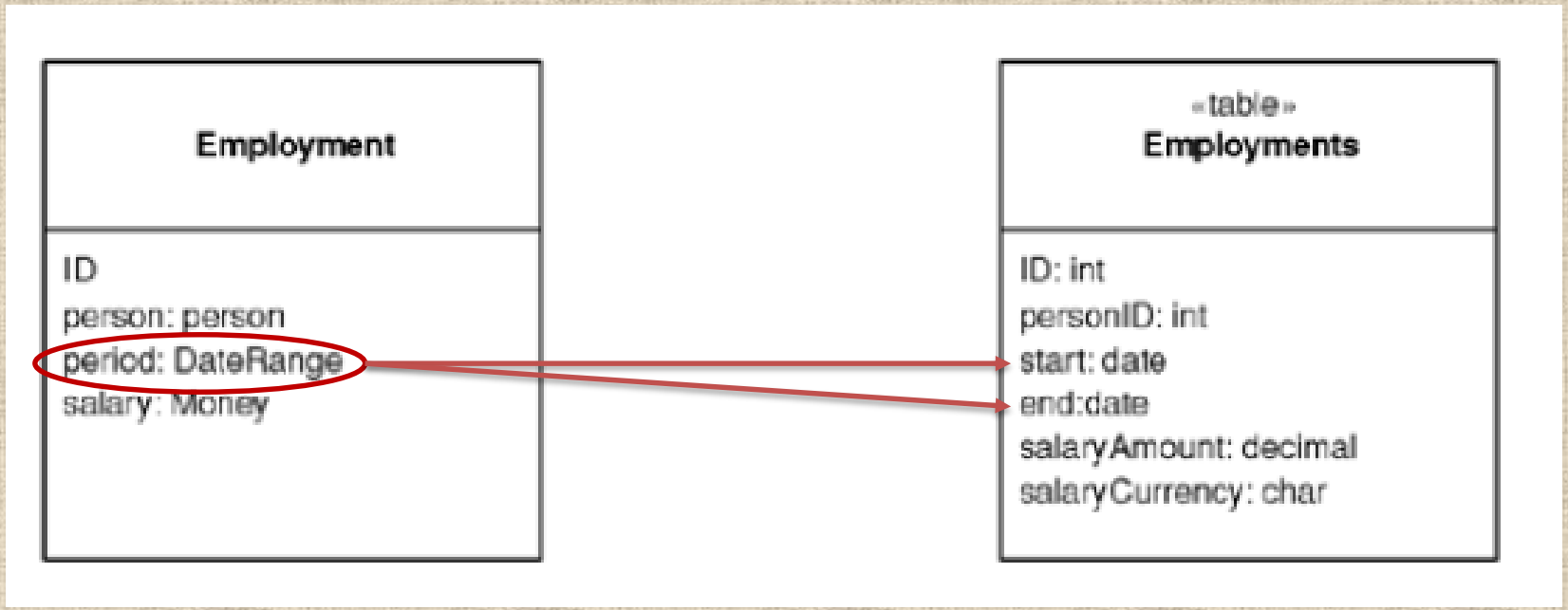
Как это работает:

Основная идея - один класс (зависимый объект) передает другому классу (владельцу) все свои полномочия по взаимодействию с базой данных. При этом у каждого зависимого объекта должен быть один и только один владелец.

В большинстве случаев вместе с объектом-владельцем загружаются и его зависимые объекты. Если загрузка зависимых объектов связана с большими расходами, а сами они используются нечасто, можно применить загрузку по требованию (Lazy Load), чтобы извлекать зависимые объекты только тогда, когда они понадобятся.

Зависимый объект может быть владельцем другого зависимого объекта. В этом случае владелец первого зависимого объекта отвечает и за второй зависимый объект. Вообще говоря, может быть целая иерархия зависимых объектов, контролируемых единственным первичным владельцем.

Типовое решение «Внедренное значение» (Embedded Value)



Отображает объект на несколько полей таблицы, соответствующей другому объекту.

В приложениях часто встречаются небольшие объекты, которые имеют смысл в объектной модели, однако совершенно бесполезны в качестве таблиц базы данных. Хорошим примером таких объектов являются денежные значения в определенной валюте или диапазоны дат.

Типовое решение внедренное значение отображает значения полей объекта на поля записи его владельца

Внедренное значение

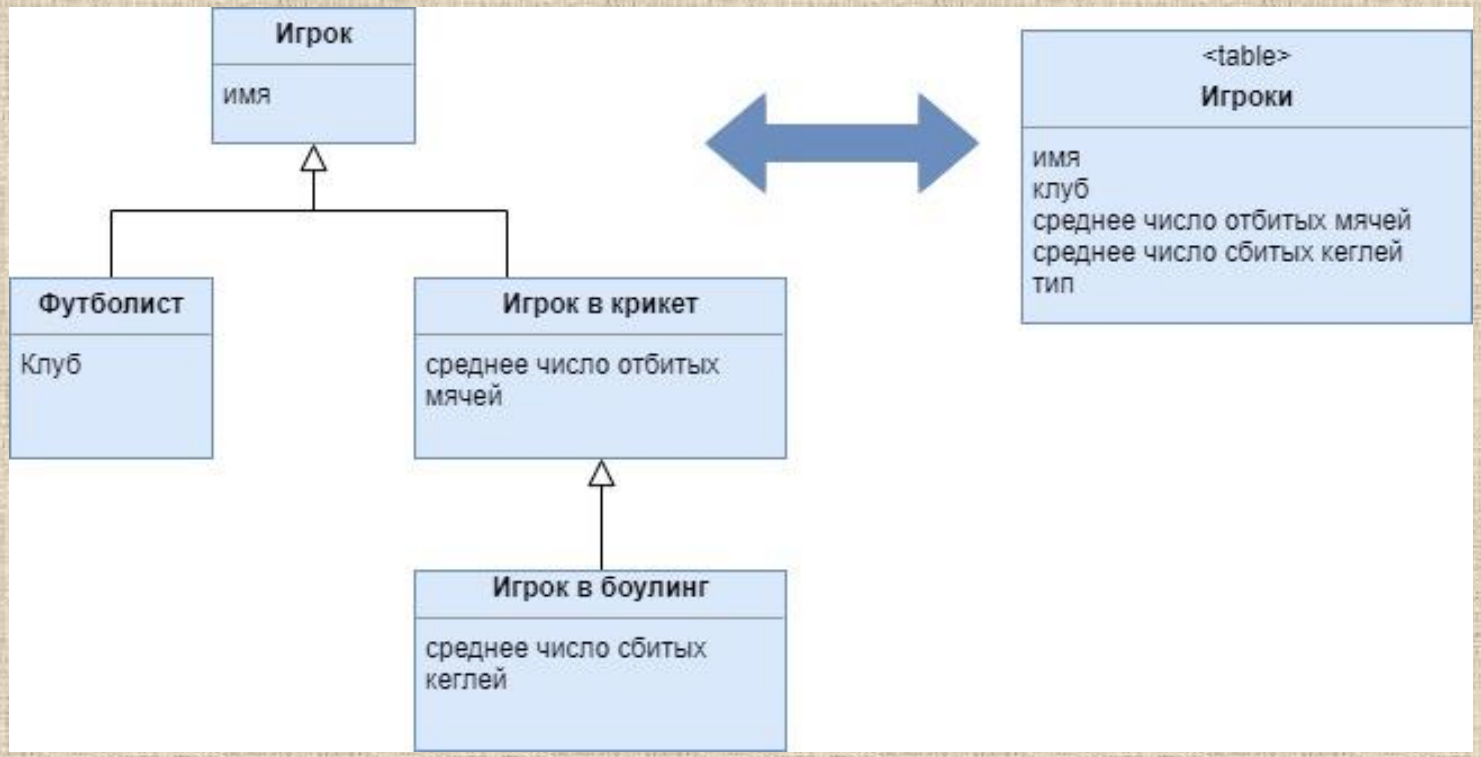
Как это работает:

Когда объект-владелец (должность) загружается в базу данных или сохраняется в ней, вместе с ним загружаются или сохраняются и зависимые объекты (диапазон дат и денежное значение). Зависимые классы не имеют собственных методов загрузки и сохранения, поскольку все эти операции выполняются их владельцем.

Главные сомнения возникают относительно того, стоит ли использовать внедренное значение для хранения полноценных объектов, связанных ссылками, например заказа и транспортной накладной. Здесь важно решить, имеет ли транспортная накладная какой-либо смысл вне контекста соответствующего заказа, в частности в плане загрузки и сохранения. Если сведения о транспортировке загружаются в память только вместе с заказом, стоит подумать о том, чтобы хранить их в одной таблице.

Еще один спорный момент связан с тем, нужно ли осуществлять доступ к транспортным накладным отдельно от заказов. Это может быть крайне важно, если создаются отчеты посредством SQL-запросов и не используется для этого отдельной базы данных.

Типовое решение «Наследование с одной таблицей» (Single Table Inheritance)



Представляет иерархию наследования классов в виде одной таблицы, столбцы которой соответствуют всем полям классов, входящих в иерархию.
Важно минимизировать количество соединений, которое возрастает при попытке отображения структуры наследования на разные таблицы.

Наследование с одной таблицей

Как это работает:

Каждому классу (а точнее, его экземпляру) соответствует одна строка таблицы; при этом поля таблицы, которых нет в данном классе, остаются пустыми .

Выполняя загрузку объекта в память, необходимо знать, в экземпляр какого класса следует поместить загружаемые данные. Для этого к таблице добавляется специальное поле, указывающее на то, экземпляр какого класса должен быть создан для загрузки данного объекта.

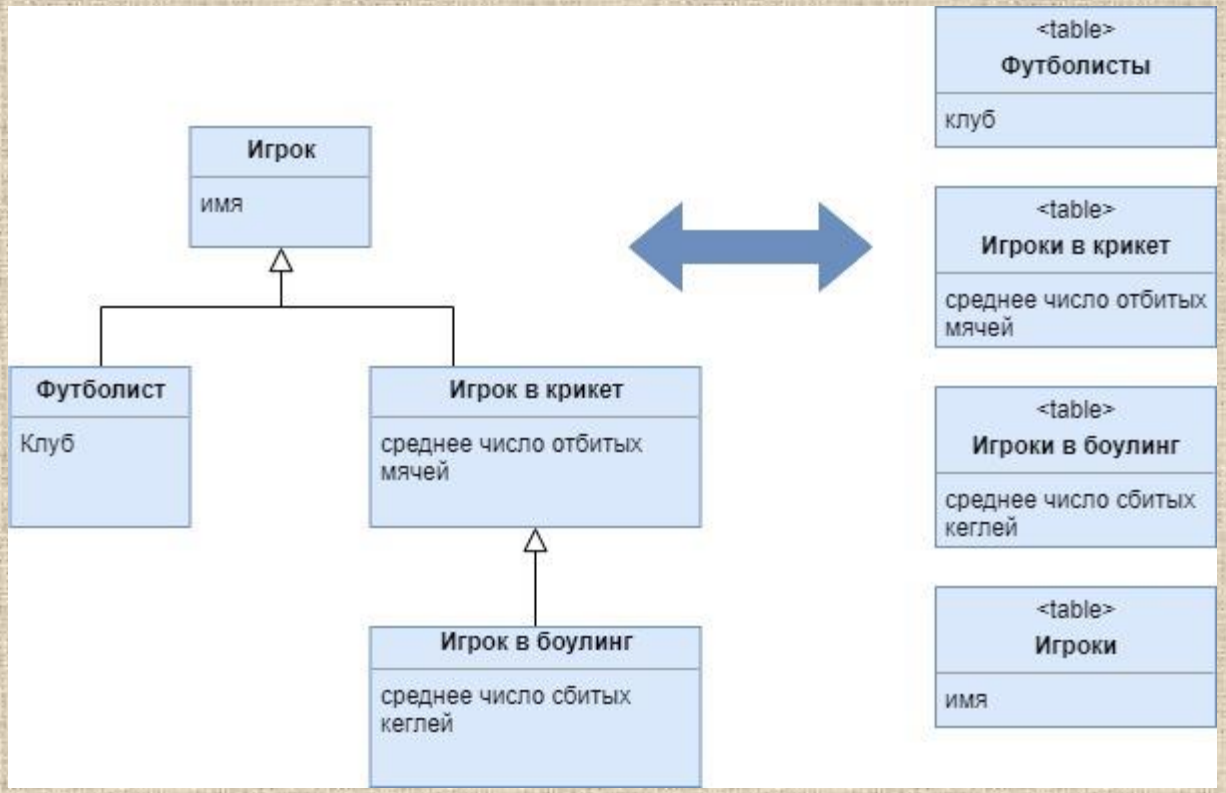
Преимущества:

- ✓ В структуру базы данных добавляется только одна таблица.
- ✓ Для извлечения данных не нужно выполнять соединение таблиц.
- ✓ Перемещение полей в производный класс или суперкласс не требует внесения изменений в структуру базы данных

Недостаток:

- ✓ Не все поля соответствуют содержимому каждого конкретного объекта, что может приводить в замешательство людей, работающих только с таблицами.
- ✓ Некоторые столбцы используются только одним-двумя производными классами, что приводит к бессмысленной трате свободного места. Критичность данной проблемы зависит от характеристик конкретных данных, а также от того, насколько хорошо сжимаются пустые поля.
- ✓ Полученная таблица может оказаться слишком большой, с множеством индексов и частыми блокировками, что будет оказывать негативное влияние на производительность базы данных.
- ✓ Все имена столбцов таблицы принадлежат единому пространству имен, поэтому необходимо следить за тем, чтобы у полей разных классов не было одинаковых имен.

Типовое решение «Наследование с таблицами для каждого класса» (Class Table Inheritance)



Представляет иерархию наследования классов, используя по одной таблице для каждого класса.

Требуется создать такую структуру базы данных, которая бы хорошо отображалась на объекты и сохраняла все возможные связи объектной модели.

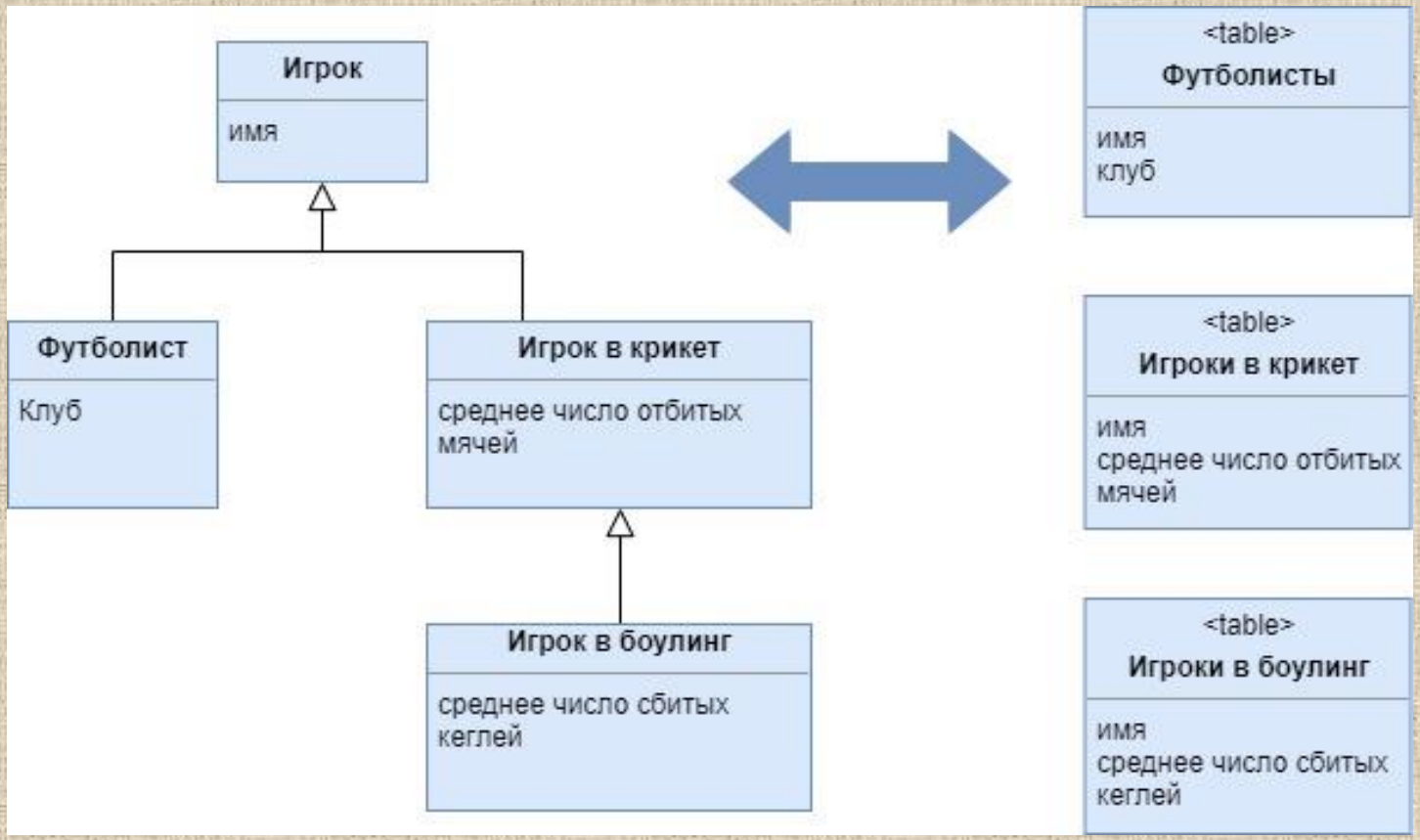
Наследование с таблицами для каждого класса

Как это работает:
Каждому классу модели предметной области соответствует своя таблица базы данных. Поля класса домена отображаются непосредственно на столбцы соответствующей таблицы.
При использовании наследования с таблицами для каждого класса возникает вопрос, как связать соответствующие строки таблиц базы данных. Возможным решением может стать использование общего значения первичного ключа, которые должны быть уникальными в пределах всех таблиц производных классов.

Преимущества:
- ✓ Все поля таблицы соответствуют содержимому каждой ее строки (т.е. есть в каждом описываемом объекте), поэтому таблицы легки в понимании и не занимают лишнего места.
- ✓ Взаимосвязь между моделью домена и схемой базы данных проста и понятна.

Недостатки:
- ✓ Загрузка объекта охватывает сразу несколько таблиц, что требует их соединения либо множества обращений к базе данных с последующим "сшиванием" результатов в памяти.
- ✓ Перемещение полей в производный класс или суперкласс требует изменения структуры базы данных.
- ✓ Таблицы суперклассов могут стать "узким местом" в вопросах производительности, поскольку доступ к таким таблицам будет осуществляться слишком часто.

Типовое решение «Наследование с таблицами для каждого конкретного класса» (Concrete Table Inheritance)



Представляет иерархию наследования классов, используя по одной таблице для каждого конкретного класса этой иерархии.

Наследование с таблицами для каждого конкретного класса

Как это работает:
Данное типовое решение подразумевает создание отдельной таблицы для каждого конкретного класса иерархии наследования. При этом каждая таблица содержит столбцы, соответствующие полям конкретного класса и всех его "предков", а потому поля суперкласса дублируются во всех таблицах его производных классов.

Преимущества:
- ✓ Каждая таблица является замкнутой и не содержит ненужных полей, вследствие чего ее удобно использовать в других приложениях, не работающих с объектами.
- ✓ При считывании данных посредством конкретных преобразователей не нужно выполнять соединений.
- ✓ Доступ к таблице осуществляется только в случае доступа к конкретному классу, что позволяет распределить нагрузку по всей базе данных.

Недостатки:
- ✓ Если в суперклассе будет изменено какое-нибудь поле, понадобится изменить каждую таблицу, имеющую данное поле, поскольку поля суперкласса дублируются во всех таблицах его производных классов.
- ✓ Отсутствует возможность моделировать отношения между абстрактными классами.
- ✓ Если поля классов домена перемещаются в суперклассы или производные классы, придется вносить изменения в определения таблиц.

Типовые решения объектно-реляционного отображения

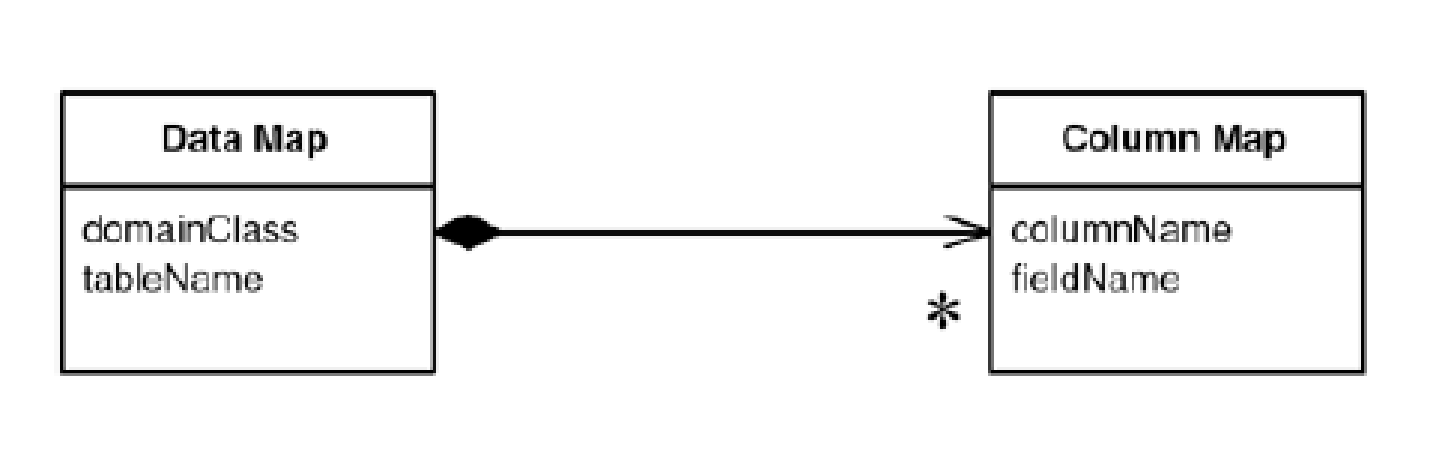
Отображение метаданных (Metadata Mapping)

Объект запроса (Query Object)

Хранилище (Repository)

Типовое решение «Отображение метаданных» (Metadata Mapping)

Хранит описание деталей объектно-реляционного отображения в виде метаданных



Типовое решение отображение метаданных позволяет разработчикам описать отображение в виде простой таблицы соответствий, которая затем может быть использована универсальным кодом для получения деталей относительно считывания, вставки и обновления данных.

Отображение метаданных

Как это работает:

Два варианта использования метаданных кодом приложения:

1. **Генерация кода (code generation)**. Создается универсальная программа, которая на вход принимает описание метаданных, а на выходе выдает исходный код классов, выполняющих отображение.
2. **Отражение (reflection).** Рефлективная программа просматривает метаданные объекта, находит метод setName и выполняет его с соответствующим аргументом. Поскольку методы (и поля) воспринимаются программой как обычные данные, она может считывать имена полей и методов из файла с метаданными и затем использовать их для отображения.

Генерация кода представляет собой гораздо менее динамичный подход, поскольку любые изменения в отображении требуют перекомпиляции и нового развертывания, как минимум, этой части программного обеспечения. А применение метода отражения позволяет просто изменить файл с описанием отображения, и существующие классы будут использовать новые метаданные.

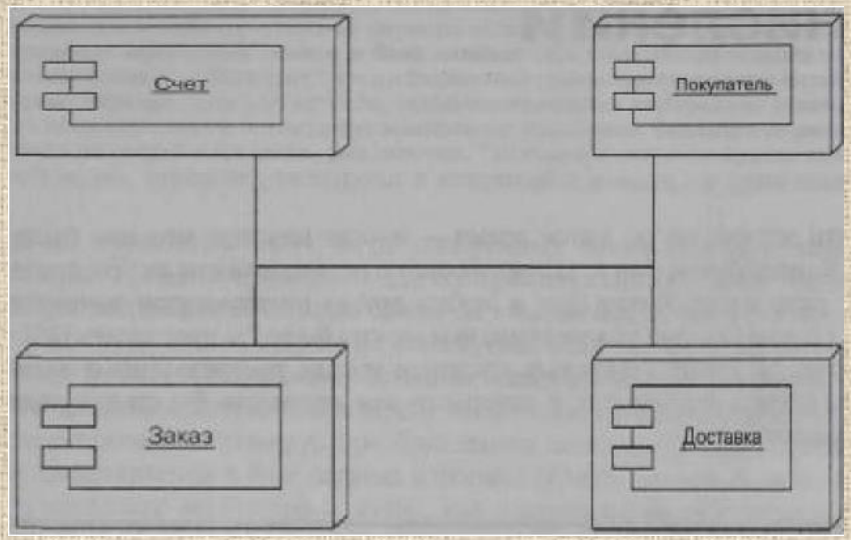
Типовые решения распределенной обработки данных

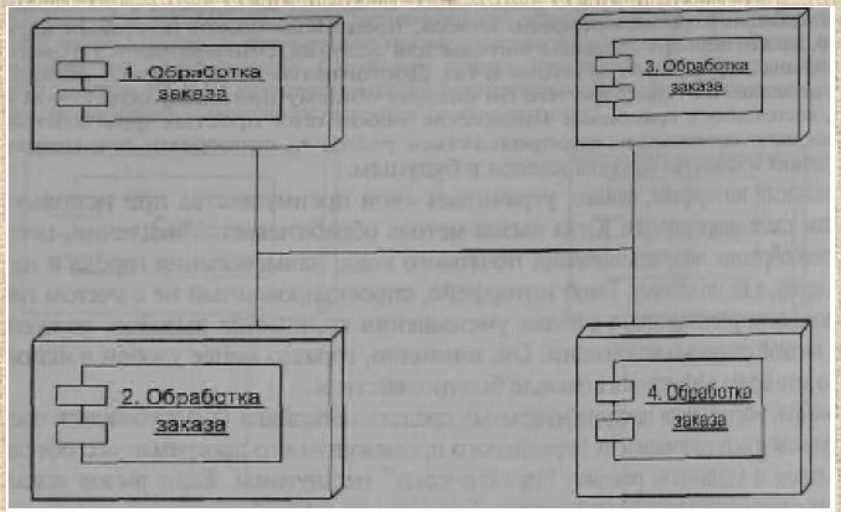
Интерфейс удаленного доступа
(Remote Facade)

Объект переноса данных
(Data Transfer Object)

Стратегии распределенных вычислений

- Соблазны модели распределенных объектов : распределение объектов путем разнесения всех компонентов приложения по разным узлам (не рекомендуется!)
- Локальные вызовы – быстрые, межпроцессные и удаленные вызовы - медленные
- Первый закон распределения объектов: «Не распределяйте объекты»

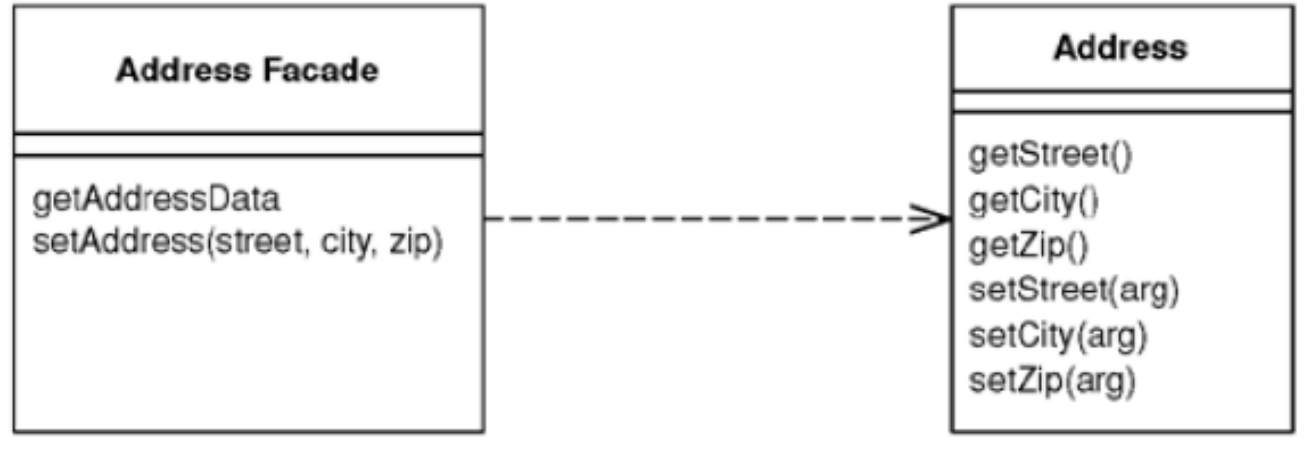




- Когда без распределения не обойтись:
  - ✓ Клиент/сервер
  - ✓ Сервер приложений / сервер БД
  - ✓ Web сервер / сервер приложений
  - ✓ Различные производители
- Сужать границы распределения
- Особое внимание удаленным интерфейсам

Типовое решение «Интерфейс удаленного доступа» (Remote Facade)

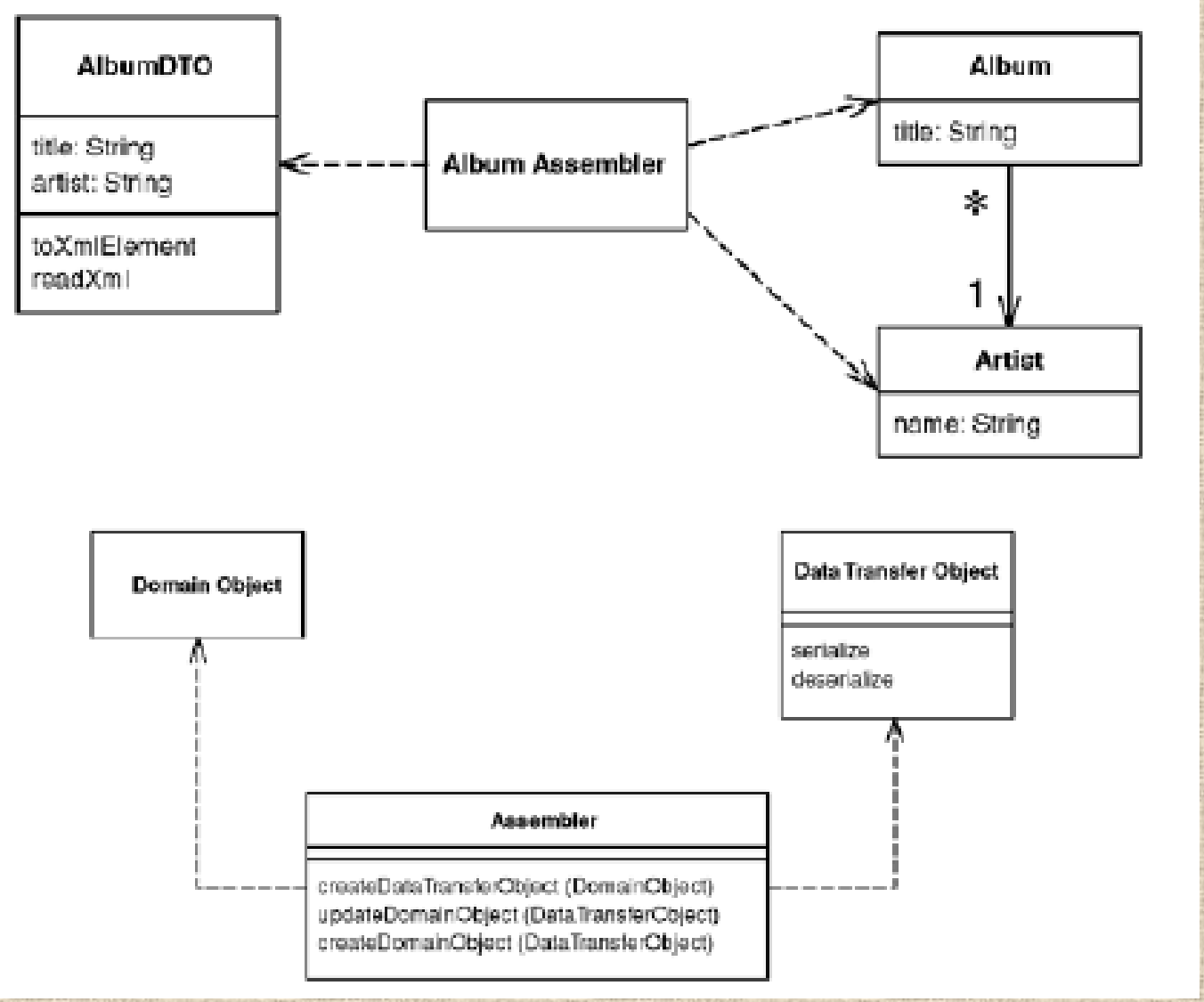
Предоставляет интерфейс с низкой степенью детализации для доступа к объектам, имеющим интерфейс с высокой степенью детализации, в целях повышения эффективности работы в сети



В объектно-ориентированной модели удобно работать с небольшими объектами, имеющими множество небольших методов. Одним из последствий высокой степени детализации интерфейса является наличие большого количества взаимодействий между объектами и, как результат, большого количества вызовов методов. Интерфейс удаленного доступа— это интерфейс с низкой степенью детализации, предоставляющий доступ к сети объектов с высокой степенью детализации. Такие объекты не имеют удаленных интерфейсов, а интерфейс удаленного доступа не содержит логики домена и лишь преобразует вызовы методов с низкой степенью детализации в последовательности методов с высокой степенью детализации.

Типовое решение «Объект переноса данных» (Data Transfer Object)

Применяется для переноса данных между процессами в целях уменьшения количества вызовов



Объект переноса данных

Как это работает:

Каждое обращение к интерфейсу удаленного доступа (Remote Facade) связано с большими затратами. Поэтому клиенту необходимо минимизировать число удаленных вызовов, а значит, каждый вызов должен возвращать как можно больше информации. Возможное решение этой проблемы - воспользоваться объектом переноса данных, который будет содержать в себе все данные, возвращаемые клиенту за один вызов. Разумеется, чтобы такой объект мог быть передан по сети, он должен поддерживать возможность сериализации. Как правило, для перемещения данных между объектом переноса данных и объектами домена применяется объект-сборщик, расположенный на стороне сервера.

Когда удаленному клиенту нужны какие-либо данные, он обращается к необходимому объекту переноса данных. Последний, как правило, содержит намного больше данных, чем было запрошено удаленным клиентом. Объект переноса данных должен содержать в себе все данные, которые могут понадобиться удаленному клиенту через какое-то время.