УДК 621.311

ОРГАНИЗАЦИЯ ВЗАИМОДЕЙСТВИЯ С ХРАНИЛИЩЕМ ДАННЫХ

Прибыльская Н.М., Шамшур А. А., Шумилин И. Н.
Белорусский национальный технический университет, г. Минск

В данной статье мы хотели бы показать, каким образом использование паттерна DAO (Data Access Object) решает большинство проблем, возникающих при работе с разного рода хранилищами данных. Любое web-приложение сегодня абсолютно немыслимо без использования баз данных и наше не исключение. В момент проектирования приложения мы столкнулись с проблемой взаимодействия бизнес-логики приложения и, собственно, хранилищем данных. По ходу проектирования, было замечено, что в слоях бизнес-логики и представления появлялись элементы, отвечающие за обмен данными между приложением и базой данных. Возникла необходимость: функционал, работающий с базой данных вынести в отдельный, независимый слой. Лучше всего решает эту проблему паттерн DAO.

DAO используется для абстрагирования и инкапсулирования доступа к источнику данных. DAO управляет соединением с источником данных для получения и записи данных. DAO реализует необходимый для работы с источником данных механизм доступа. Источником данных может быть персистентное хранилище, внешняя служба, репозиторий, или бизнес-служба. Использующие DAO бизнес-компоненты работают с более простым интерфейсом, предоставляемым объектом DAO своим клиентам. DAO полностью скрывает детали реализации источника данных от клиентов. Поскольку при изменениях реализации источника данных представляемый DAO интерфейс не изменяется, этот паттерн дает возможность DAO принимать различные схемы хранилищ без влияния на клиенты или бизнес-компоненты. По существу, DAO выполняет функцию адаптера между компонентом и источником данных.

Разрешает прозрачность. Бизнес-объекты могут использовать источник данных, не имея знаний о конкретных деталях его реализации. Доступ является прозрачным, поскольку детали реализации скрыты внутри DAO.

Облегчает миграцию. Уровень объектов DAO облегчает приложению миграцию на другую реализацию базы данных. Бизнес-объекты не знают о деталях реализации используемых данных. Следовательно, процесс миграции требует изменений только в уровне DAO. Более того, при использовании стратегии генератора можно предоставить конкретную реализацию генератора для каждой реализации хранилища данных. В этом случае миграция на другую реализацию хранилища означает предоставление приложению новой реализации генератора.

Уменьшает сложность кода в бизнес-объектах. Поскольку объекты DAO управляют всеми сложностями доступа к данным, упрощается код бизнес-компонентов и других клиентов данных, использующих DAO. Весь зависящий от реализации код (например, SQL-команды) содержится в DAO, а не в бизнес-объекте. Это улучшает читаемость кода и производительность разработки.

Централизует весь доступ к данным в отдельном уровне. Поскольку все операции доступа к данным реализованы в объектах DAO, отдельный уровень доступа к данным может рассматриваться как уровень, изолирующий остальную часть приложения от реализации доступа к данным. Такая централизация облегчает поддержку и управление приложением.

Бесполезна для управляемой контейнером персистенции, поскольку EJB-контейнер управляет компонентами с управляемой контейнером персистенцией (CMP – container-managed persistence), контейнер автоматически обслуживает весь доступ к хранилищу данных. Приложения, использующие компоненты управления данными этого типа, не нуждаются в уровне объектов DAO, поскольку сервер приложений обеспечивает эту функциональность. Однако, объекты DAO остаются полезны в случаях, когда необходимо использовать комбинацию CMP (для компонентов управления данными) и BMP (для сессионных компонентов, сервлетов).

Добавляет дополнительный уровень. Объекты DAO создают дополнительный уровень объектов между клиентом данных и источником данных, который должен быть разработан и реализован для использования преимуществ, предлагаемых данным паттерном. Но за реализуемые при этом преимущества приходится платить дополнительными усилиями при разработке. [3]

Требуется разработка иерархии классов. Мы реализовали эту стратегию с паттерном Factory Method, а затем перешли к паттерну Abstract Factory.

У нас архитектура слоя построена следующим образом. На вершине иерархии находится абстрактный класс, в который вынесены основные, повторяющиеся действия по подключению и отправке запроса к базе данных. Затем было проведено разбиение на подслой отвечающий за обработку ошибок, подслой всех действий, которые можно осуществлять через приложение касательно области базы данных (регистрация/логинация и т.п.) и собственно имплементацию всего описанного.

В целом, благодаря такому подходу, а именно введению слоя DAO и использованию на нем JDBC, позволило сократить время разработки сегмента приложения, отвечающего за взаимодействие с базой данных.

Такое решение позволило существенно сократить время необходимое на расширение, модификацию и локализацию ошибок. На рисунке 1

наглядно показан прирост в выигрыше временных затрат на некоторые действия.

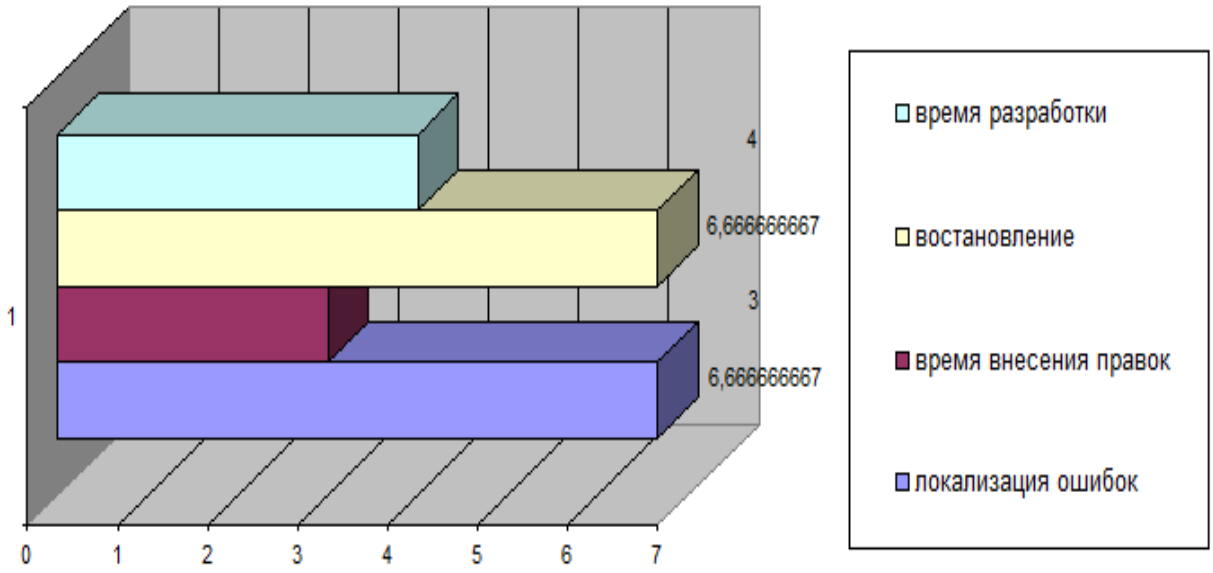

Рисунок 1 – Выигрыш временных затрат на разработку и сопровождение проекта

Отсюда видно, что применение такого решения в проектах положительно сказывается на качестве конечного продукта, а также на возможности по его расширению и сопровождению.

1. Серверные приложения на языке Java. Р. Р. Мухамедзянов. Издательство: «СОЛОН – Р», 2002.
2. Технологии программирования на Java 2. Книга 2. Распределенные приложения. Х. М. Дейтел, П. Дж. Дейтел, С. И. Сантри. Издательство: «Бином-Пресс», 2009 г.
3. JavaTutor [Электронный ресурс]. – Режим доступа: http:/www.javatutor.net – Дата доступа: 24.11.2017.