Command Query Responsibility Segregation (CQRS)

Перевод статьи: From CQS to CQRS (Herberto Graca)

Когда у нас есть ориентированое на данные приложение (data-centric), т. е. приложение, реализующее только основные операции CRUD и оставляющее бизнес-процессы (например, какие данные изменять и в каком порядке) в ответственности пользователя, у нас есть преимущество в том, что пользователи могут изменить бизнес-процесс без необходимости изменения приложения. С другой стороны, это означает, что все пользователи должны знать все детали всех бизнес-процессов, которые могут быть выполнены с помощью приложения, что является большой проблемой, когда у нас есть нетривиальные процессы и много людей, которые должны их знать.

Data-centric приложение ничего не знает о бизнес-процессах, т.е. модель предметной области (если ее можно так называть в данном случае) не отражает никаких процессов предметной области, кроме изменения данных. Это становится *прославленной абстракцией модели данных*. Процессы существуют только в головах пользователей приложения, или даже на наклейках-напоминалках, приклееных на экран компьютера.

С другой стороны нетривиальное и действительно полезное приложение направлено на то, чтобы снять бремя “процесса” с плеч пользователя, понимать его намерения и обрабатывать поведение, а не просто хранить данные.

Разделение команды и запроса (CQS)

Как утверждает Мартин Фаулер, термин command query separation был придуман Бертраном Мейером в его книге «Object Oriented Software Construction» (1988) - книге, которая, как говорят, является одной из самых влиятельных книг во времена становления ООП.

Мейер утверждает, что у нас **не должно быть методов, которые одновременно изменяют и возвращают данные**. Таким образом, у нас может быть два типа методов:

1. **Queries** (запросы): возвращают данные, но не изменяют их, и как следствие, не имеют побочных эффектов;
2. **Commands** (команды): изменют данные, но не возвращают их.

Это является отражением принципа единственной ответственности (SRP) на уровне методов.

Однако, есть некоторые шаблоны, которые являются исключениями из этого правила. Как, опять же, говорит Мартин Фаулер, традиционные реализации очереди и стека содержат, например, метод `.pop()`, который удаляет элемент из стэка/очереди (т.е. изменяет состояние стэка/очереди) и возвращает это элемент.

Шаблон Command

Основной идеей шаблона Command является переход от data-centric приложения к process-centric приложению, т.е. приложению которое содержит в себе логику предметной области и бизнес-процессов.

На практике это означает, что вместо того, чтобы заставлять пользователя выполнять действие `CreateUser` действий, за которым ему следует выполнить действия `ActivateUser` и `SendUserCreatedEmail`, пользователю будет необходимо

просто выполнить команду `RegisterUser`, которая будет выполнять вышеупомянутую последовательность действия `CreateUser`, `ActivateUser`, `SendUserCreatedEmail` как единый бизнес-процесс.

Более интересным может быть пример, когда у нас есть форма для изменения данных клиента. Допустим, форма позволяет нам изменить имя клиента, адрес, номер телефона и выставлять статус клиента (является ли он привелегированным или нет). Клиент может быть привелегированным, только если он оплатил свои счета.

В приложении CRUD мы получаем данные и проверяем, оплатил ли клиент свои счета, и далее принимаем или отклоняем запрос на изменение данных. Однако у нас здесь фигурирует два разных бизнес-процесса: изменение имени клиента, адреса и номера телефона должно быть успешным, даже если клиент не оплатил свои счета и не является привелегированным.

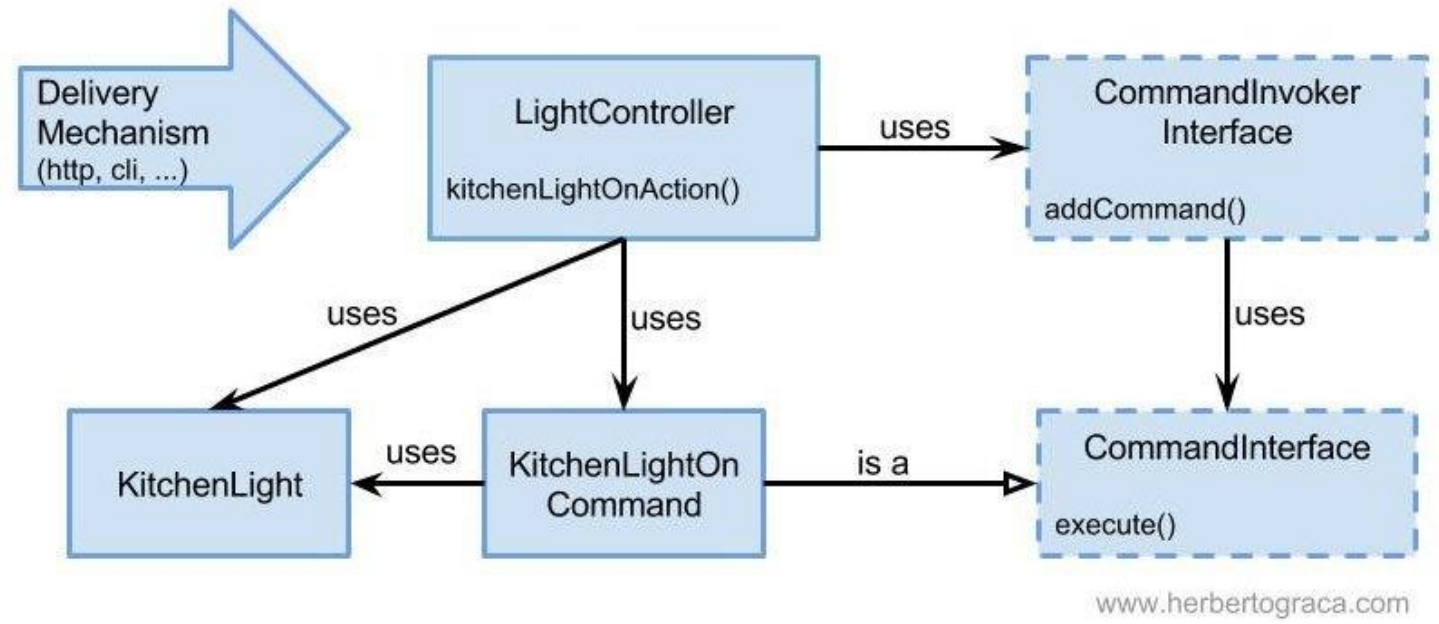
Используя шаблон Command, мы можем сделать эти различия явными в коде, создав две команды, представляющие два разных бизнес-процесса: одну для изменения данных клиента, а другую для выставления статуса клиента.

Однако необходимо помнить, что это не означает, что не может быть простой команды `CreateUser`. Т.е. использование CRUD может прекрасно сосуществовать с случаями использования Command, которые представляют собой сложный бизнес-процесс, но очень важно не ошибиться в выборе между этими подходами.

Технически шаблон Command, как описано в книге «Head First Design Patterns», инкапсулирут все, что необходимо для выполнения некоторых действий или последовательности действий. Это особенно полезно, когда у нас есть несколько различных бизнес-процессов (Command), которые должны выполняться из одного и того же места, т.е. классы Command должны реализовывать один и тот же интерфейс. Например, все команды будут иметь один и тот же метод `.execute()`. Это позволит поставить любой бизнес-процесс (Command) в очередь и выполнить тогда, когда это возможно, синхронно или асинхронно.

В книге «Head First Design Patterns» приведен пример пульта дистанционного управления для управления освещением в доме. Я буду использовать тот же пример.

Итак, давайте предположим, что у нас есть пульт дистанционного управления освещением в доме, и на нем у нас есть 2 кнопки: одна для включения и вторая для включения освещения на кухне. Каждая из этих кнопок представляет собой команду, которую мы можем отдать системе освещения. Вот какой вид может иметь такая система:



Это конечно наивный дизайн, который даже не использовал правильный UML. Но я надеюсь, что он служит своей цели, поэтому давайте рассмотрим эту диаграмму: в качестве реакции на сигнал от некоторого механизма доставки (http request и т.п.), создается `LightController`, в который внедрен `CommandInvoker` через аргумент конструктора и далее вызывается определенное действие этого контроллера (метод `.kitchenLightOnAction()`). Это действие будет инстанцировать соответствующий свет `KitchenLight`, а также будет инстанцировать соответствующую Command'у `KitchenLightOnCommand`, передавая ей в качестве аргумента конструктора `KitchenLight`. Затем `KitchenLightOnCommand` передается в `CommandInvoker`, который будет выполнять ее синхронно или асинхронно.

Аналогично, чтобы выключить свет, мы создадим другую Command'у, но дизайн останется таким же.

Итак, у нас есть Command'ы, чтобы включить и выключить свет. Но что делать, если нужно установить освещение на 50% мощности? Мы создаем другую Command'у! А если нам нужно установить освещение на 25% и 75%? Мы создаем больше Command! Что делать, если вместо кнопок у нас есть диммер, который установит свет практически на любое значение?! Мы не можем создавать Command'ы бесконечно!!!

Проблема реализации, на данном этапе, состоит в том, что логика Command будет одинаковой, но данные (процент мощности освещения) будут отличаться каждый раз. Таким образом, мы должны создать Command'у, которая имеет ту же логику и просто выполняется с разными данными. Но тогда у нас есть проблема с интерфейсом – метод `.execute()` не принимает аргументы. Если заставить его принимать аргументы, то это нарушило бы всю техническую идею шаблона Command, которая подразумевает инкапсуляцию всего, что необходимо для выполнения некоторого бизнес-процесса, чтобы код, запускающий Command'у на выполнение, не знал о том, что конкретно будет выполняться.

Command Bus

Чтобы обойти упомянутое ограничение шаблона Command необходимо применить один из старейших принципов ООП: **необходимо отделять то, что изменяется от того, что не изменяется**.

В нашем случае данные – это то, что изменяется, а логика Command'ы – это то, что не изменяется, поэтому мы можем разделить их на два класса. Один будет простой DTO для хранения данных и мы будем называть этот DTO **Command'ой**. А другой будет содержать логику для выполнения и мы будем

называть его **Handler'ом**, который будет иметь метод для выполнения необходимой логики, например:

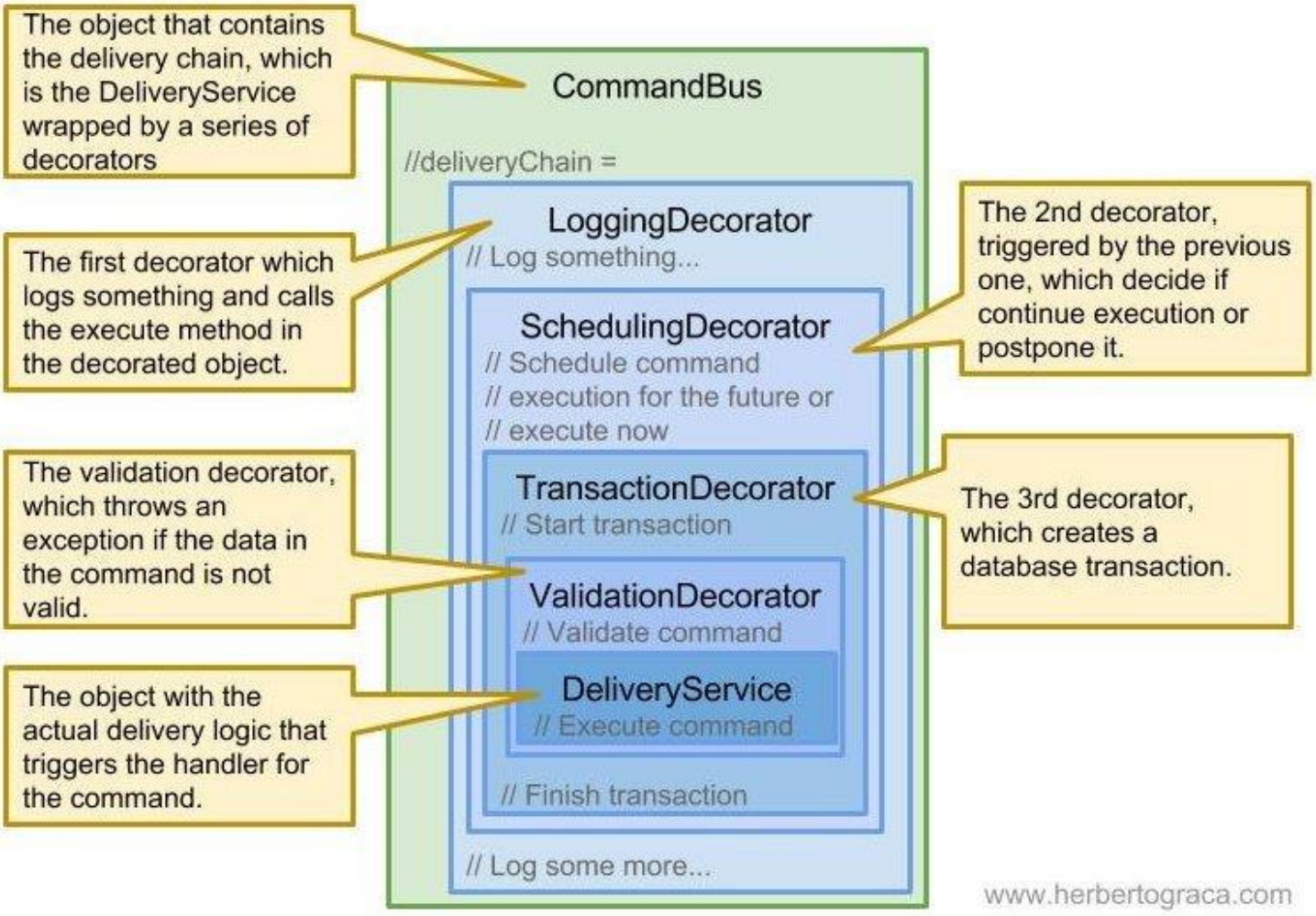
```
void execute(CommandInterface command);
```

Мы также преобразуем `CommandInvoker` в нечто, способное получить Command'у и отыскать Handler, который должен обрабатывать эту Command'у. Т.е. `CommandInvoker` превращается в Command Bus.

Кроме того, как часто бывает, многие Command'ы не должны обрабатываться немедленно, они могут быть поставлены в очередь и выполнены позже, асинхронно. Это имеет некоторые преимущества, которые делают систему более надежной:

- Ответ пользователю возвращается быстрее, т.к. мы не обрабатываем Command'у немедленно;
- Если из-за системного дефекта, например, bug'а или недоступности БД, Command'а не выполнится сразу, то пользователь этого даже не почувствует. А Command'а будет выполнена, когда системный дефект будет устранен.

Наличие централизованного места (Command Bus), где мы запускаем Handler'ы, также дает нам преимущество иметь одно место, где мы можем добавить логику, которая будет выполняться для всех Handler'ов, до и/или после выполнения Handler'а. Например, мы можем проверить данные Command'ы перед передачей её подходящему Handler'у, или мы можем обернуть выполнение Handler'а вокруг транзакции БД, или мы можем настроить Command Bus для поддержки сложных операций организации очереди и асинхронного выполнения Command. Обычно для реализации подобных дополнительных возможностей используется шаблон **Decorator**.

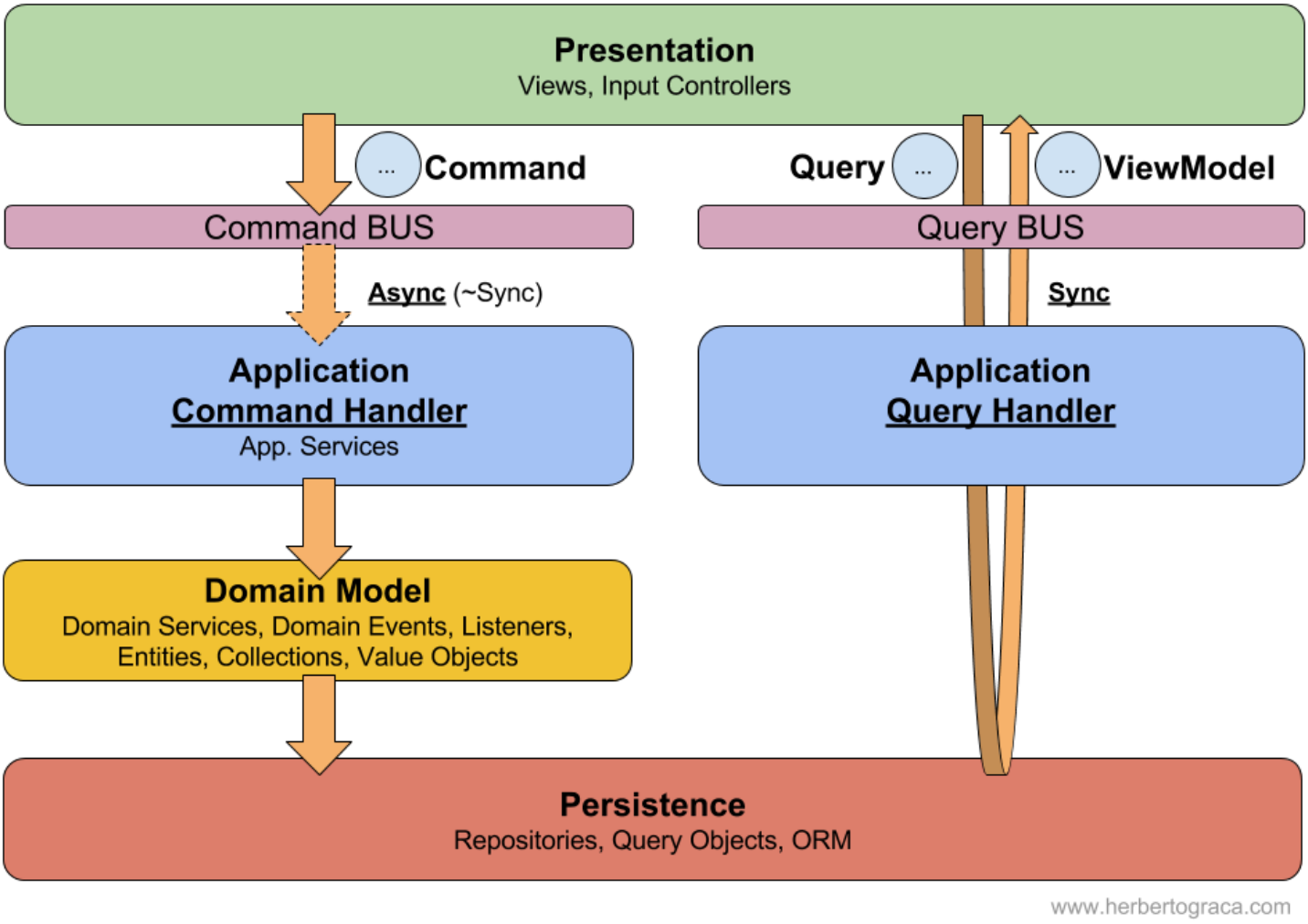


Это позволяет нам создавать наши собственные Decorator'ы и настраивать Command Bus (возможно, стороннего производителя), который будет обернут в эти Decorator'ы в необходимом порядке, для добавления необходимой функциональности. Если нам нужна очередь Command, то мы добавим Decorator для управления очередью Command. Если мы не используем транзакционную БД, то нам не нужен Decorator, чтобы обернуть выполнение Handler'а вокруг транзакции БД и т.д.

Command Query Responsibility Segregation (CQRS)

CQRS получается при объединении следующих концепций и понятий: CQS, Command, Query, Command Bus. CQRS подход может быть реализован различными способами и на разных уровнях, возможно, только на стороне

Command'ы, или, может быть, без использования Command Bus'а. Ниже можно видеть диаграмму, которая отражает то, как я вижу полную реализацию CQRS:



Query side

В соответствии с CQS сторона Query будет только возвращать данные, не изменяя их. Поскольку в данном случае нам не нужно выполнять бизнес-логику над этими данными, то нам не нужны бизнес-объекты (т. е. Entity и т.п.), и также нам не нужен ORM для получения Entity'ей. Т.е. нам просто нужно запросить данные, которые нам необходимы для формирования представления, которое должно быть показан пользователю!

Это способствует повышению производительности, т.е. при запросе данных нам не нужно проходить уровни бизнес-логики, чтобы получить необходимые

данные. Вместо этого мы получаем необходимые данные напрямую из Persistence слоя.

Благодаря такому разделению становится возможной еще одна оптимизация, которая заключается в полном разделении хранилища данных на два отдельных хранилища: одно оптимизировано для записи, а другое оптимизировано для чтения. Например, если мы используем СУБД:

- Операции чтения не требуют проверки целостности данных, ей вообще не нужны ограничения внешнего ключа, поскольку проверка целостности данных выполняется на этапе записи этих данных в хранилище. Таким образом, мы можем удалить ограничения целостности данных из БД, которая предназначена для чтения;
- Также можно использовать DB Views с данными, которые нам нужны для формирования каждого представления, что делает запросы тривиальными и быстрыми.

Также, если у нас есть специализированное DB View данных для каждого представления, что делает запрос тривиальным, то зачем нам нужна RDBMS для наших чтений?! Может быть, мы можем использовать хранилище документов для наших чтений, такое как MongoDB или даже Redis, которое быстрее. Может быть, да, а может быть, и нет, я просто говорю, что, возможно, стоит подумать об этом, если приложение имеет проблемы с производительностью на стороне чтения.

Сам запрос может быть выполнен с помощью объекта Query, который возвращает массив данных, которые необходимы в представлении. Или мы можем использовать что-то более сложное, например, Query Bus, который, например, получает имя шаблона представления, использует объект Query для запроса данных и возвращает экземпляр ViewModel, который нужен шаблону представления.

Такой подход может решить несколько проблем, выявленных Грегом Янгом:

- *Большое количество методов чтения в репозиториях, часто также включают информацию, необходимую для paging'ации и/или сортировки;*
- *Getter'ы раскрывают внутреннее состояние объектов домена (Entity) в процессе построения DTO;*
- *Использование путей предварительной выборки в вариантах использования направленных на чтение данных;*
- *Загрузка нескольких корней агрегатов для построения DTO вызывает неоптимальные запросы к модели данных. Кроме того, границы агрегатов можно спутать из-за операций построения DTO;*
- *Однако самая большая проблема заключается в том, что оптимизация запросов чрезвычайно сложна: поскольку запросы работают с объектной моделью, а затем преобразуются в модель данных. Таким образом, в случае использования ORM, может быть очень трудно оптимизировать запросы.*

Command side

Как объяснялось ранее, с помощью Command мы преобразуем приложение от data-centric дизайна к Domain-Driven Design'у.

Удаляя операции чтения из кода, обрабатывающего Command'ы, проблемы, выявленные Грегом Янгом, просто исчезают:

- Объектам предметной области (Entity) больше не нужно выставлять на показ свое состояние;
- Репозитории имеют очень мало методов поиска, за исключением `getById`.

Отношения "one-to-many" и "many-to-many" между Entity'ми могут оказать серьезное влияние на производительность механизма ORM. Хорошей новостью

является то, что мы редко нуждаемся в таких связях при обработке Command. Такие связи в основном используются для Query'ей, а т.к. мы только что отделили обработку Query от обработки Command, то мы можем удалить эти связи между Entity'ми на стороне обработки Command. Я не говорю здесь о связях между таблицами в СУБД, эти ограничения внешнего ключа все еще должны существовать на стороне БД, я говорю о соединениях между Entity'ми, которые конфигурируются на уровне ORM.

Действительно ли нам нужна коллекция заказов в объекте клиента? В какой Command'е эта коллекция может понадобиться? На самом деле, какой Command'е могут понадобиться отношения "one-to-many" и "many-to-many"? В любом случае, большинство Command содержат только один или два идентификатора.

Udi Dahan 2009, [Clarified CQRS]

Следуя тем же мыслям, что и для стороны Query, если на стороне Command не используется сложных запросов, можем ли мы заменить RDBMS хранилищем документов или хранилищем пар ключа-значения для хранения сериализованных Entity'ей? Может быть, да, может быть, нет, я просто говорю, что, возможно, стоит подумать об этом, если у приложения есть проблемы с производительностью на стороне записи.

События бизнес-процессов

После обработки Command'ы и, если она была успешно обработана, Handler может инициировать Event, уведомляющий остальную часть приложения о том, что произошло. Event'ы должны быть названы как Command'а, которая инициировала его, за исключением того, что имя Event'а должно быть в прошедшем времени.

Заключение

С помощью CQRS мы можем полностью отделить модель чтения от модели записи, что позволяет нам оптимизировать операции чтения и записи. Это повышает производительность, сопутствует ясности и простоте кода, его поддерживаемости, повышает способность кода отражать понятия и концепции предметной области.

И опять же, этот подход используется для следования принципу единственной ответственности (SRP), повышения внутренней связности и снижения внешней зависимости.

Тем не менее, хорошо иметь в виду, что, хотя CQRS и позволяет сделать приложение очень надежным, но это не означает, что все приложения должны быть построены таким образом: мы должны использовать то, что нам нужно, когда нам это нужно.