From CQS to CQRS

hgraca Architecture, Development, Series, The Software Architecture Chronicles October 19, 2017October 19, 2017 12 Minutes

This post is part of The Software Architecture Chronicles (https://herbertograca.com/2017/07/03/the-software-architecture-chronicles/), a series of posts about Software Architecture (https://herbertograca.com/category/development/series/software-architecture/). In them, I write about what I've learned on Software Architecture, how I think of it, and how I use that knowledge. The contents of this post might make more sense if you read the previous posts in this series.

When we have an application that is data-centric, ie. implements only basic CRUD operations and leaves the business process (ie. what data to change and in what order) to the user, we have the advantage that the users can change the business process without the need to change the application. In the other hand, this implies that all users need to know all details of all business processes that can be performed using the application, which is a big problem when we have non-trivial processes and a lot of people that need to know them.

In a data-centric application, the application has no knowledge of the business processes, so the domain is unable to have any verbs, and is unable to do anything else aside from changing raw data. It becomes a *glorified abstraction of the data model*. The processes exist only in the heads of the application users, or even in post-its pinned to the computer screen.

A non-trivial, and really useful, application aims to remove the "process" burden from the user's shoulders by capturing their intentions, making it an application capable of processing behaviours as opposed to simply storing data.

CQRS is the result of an evolution of several technical concepts that work together to help provide the application with an accurate reflection of the domain, while overcoming common technical limitations.

Command Query Separation

As Martin Fowler states, *the term 'command query separation' was coined by Bertrand Meyer in his book "Object Oriented Software Construction (https://www.amazon.com/gp/product/0136291554?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0136291554)"* (1988) – a book that is said to be one of the most influential OO books, during the early days of OO.

Meier defends that, as a principle, **we should not have methods that both change data and return data**. So we have two types of methods:

1. **Queries**: Return data but do not change data, therefore having no side effects;
2. **Commands**: Change data but do not return data.

In other words, *asking a question should not change the answer and doing something should not give back an answer*, which also helps to respect the Single Responsibility Principle.

Nevertheless, there are some patterns that are exceptions to this rule as, again, Martin Fowler says, the traditional queue and stack implementations pop an element of the queue or stack, both changing the queue/stack and returning the element removed from it.

Command Pattern

The main idea of the Command Pattern is to move us away from a data-centric application and into a process-centric application, who has domain knowledge and application processes knowledge.

In practice, this means that instead of having a user execute a "CreateUser" action, followed by an "ActivateUser" action, and a "SendUserCreatedEmail" action, we would have the user simply execute a "RegisterUser" command which would execute the previously mentioned three actions as an encapsulated business process.

A more interesting example is when we have a form to change a client data. Let's say that the form allows us to change the client name, address, phone number and if he is a preferred client. Let's also say that a client can only be a preferred client if he has paid his bills. In a CRUD application, we would receive the data, check if the client has paid his bills and accept or reject the data change request. However, we have here two different business processes: Changing the client name, address and phone number should succeed even if the client hasn't paid his bills. Using the command pattern we can make this distinction clear in the code by creating two commands representing the two different business processes: one to change the client data, and another one to upgrade the client to the preferred status, both processes being triggered by the same UI screen.

> *Provide us with the right level of granularity and intent when modifying data. That's what commands are all about.*
>
> *Udi Dahan 2009, Clarified CQRS (http://udidahan.com/2009/12/09/clarified-cqrs/)*

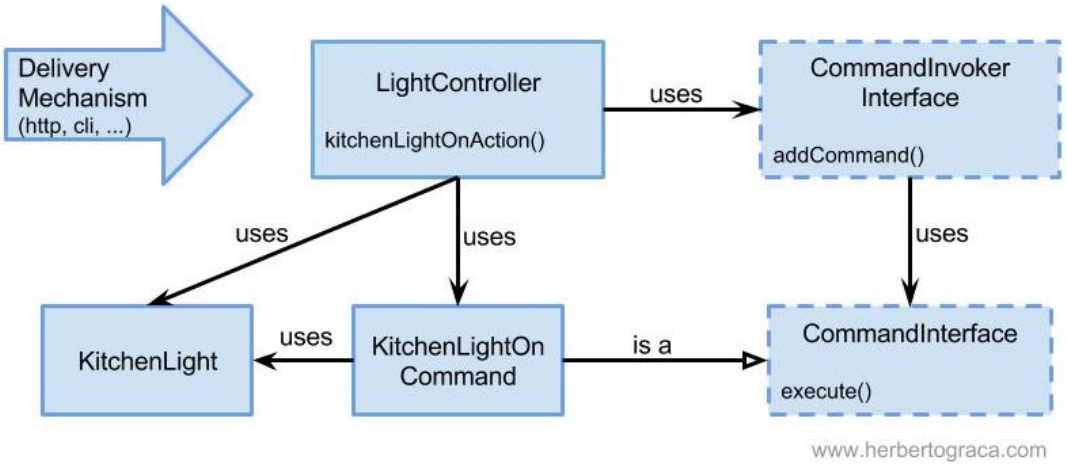
It's good to remember, however, that this does not mean that there can't be a simple "CreateUser" command. CRUD use cases can perfectly coexist with use cases that carry intent, that represent a complex business process, but it's crucial not to mistake them.

Technically, the Command Pattern, as described in Head First Design Patterns, (https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124) will encapsulate all that is needed to execute some action or sequence of actions. This is especially useful when we have several different business processes (Commands) that need to run from the same location, in the same way, so they need to have the same interface. For example, all commands will have the same method *execute()* so that, at some point, any command can be triggered independently of what command it is. This will allow for any business process (Command) to be queued and executed when possible, be it in sync or async.

In the book Head First Design Patterns (https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124), the example given is that of a remote control for lights in a house. I will go with the same example, although to pinpoint where I feel it falls short.

So, let's suppose we have a remote control for our house lights and, on it, we have a button to turn the kitchen lights on and another button to turn those lights off. Each of these buttons represents a command we can issue to the house lights system.

This is how such a system could be designed:



It's a naive design, of course, it doesn't even take into consideration a DIC and I didn't even use proper UML. But I hope it serves the purpose, so let's go through the diagram above: As a reaction to input from some delivery mechanism, the *LightController* is instantiated with a *CommandInvoker* as a constructor argument and a specific controller action is triggered, ie. the *kitchenLightOnAction*. This action will instantiate the appropriate light, ie. KitchenLight, and will also instantiate the appropriate Command, ie. KitchenLightOnCommand, passing it the light object as a constructor argument. The command is then given to the CommandInvoker which will execute it, at some point. To turn the light off we would create another Action and another Command, but the design would be essentially the same.

So we have a command to turn the lights on and another for turning them off. What if we need to set them to 50% power? We create another command! And if we need to set them to 25% and 75%? We create more commands! What if, instead of buttons we have dimmer that will set the lights to virtually any value?! We can't create infinite commands!!!

The implementation problem, at this point, is that the logic in the command will be the same, but the data (the percentage of power) will be different every time. So we should create a command that has the same logic and just executes with different data, but then we have the problem of the interface *execute()* method not accepting arguments. If it did, it would break the whole technical idea of the Command, in the first place (*encapsulate all that is needed to execute some business process without knowing exactly what is going to be executed*).

Of course, we could work around this by passing the data to the command in its constructor, but that would not be elegant. It would, in fact, be a hack because that data is not something the object needs in order to exist, it is something it needs to execute some of its logic, so that data is a dependency of the method, not a dependency of the object.

We might also be able to use native language constructs to work around this, but that would not be elegant either.

Command Bus

What we can do, to solve the mentioned Command Pattern limitation, is to apply one of the oldest principles in OO: **separate what changes from what doesn't change**.

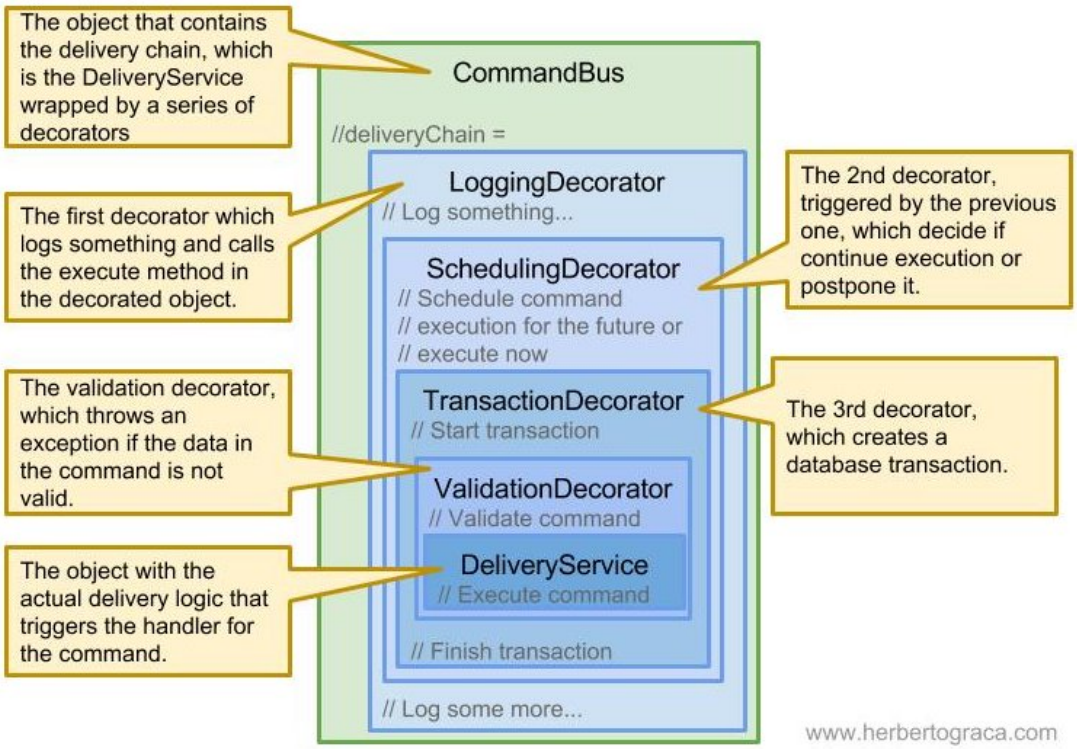
In this case what changes is the data and what doesn't change is the logic executed in the command so we can separate them into two classes. One will be a simple DTO to hold the data (we will call it **Command**), and the other will hold the logic to be executed (we will call it the **Handler**) which will have a method to trigger the logic execution ie. *execute(CommandInterface $command): void*. Também evoluiremos o Invocador de Comando para algo capaz de receber um Comando e descobrir qual manipulador pode lidar com ele. Vamos chamá-lo de **Command Bus**.

Furthermore, by changing the user interface patterns a bit, many commands don't need to be processed immediately, they can be queued and executed asynchronously. This has some advantages that make the system more robust:

- The response to the user is sent back faster because we don't process the command immediately;
- If because of system defect, like a bug or the DB being offline, a command fails, the user doesn't even realise it. The command can simply be replayed when the problem is solved.

Having a centralised place where we trigger the logic we need to run (where we trigger the handler), also gives us the advantage of having one place where we can add logic that will be executed for all the handlers, before and/or after the handler execution. For example, we can validate a command data before passing it to the handler, or we can wrap the handler execution around a DB transaction, or we can make the command bus support complex queueing operations and async command/handler execution.

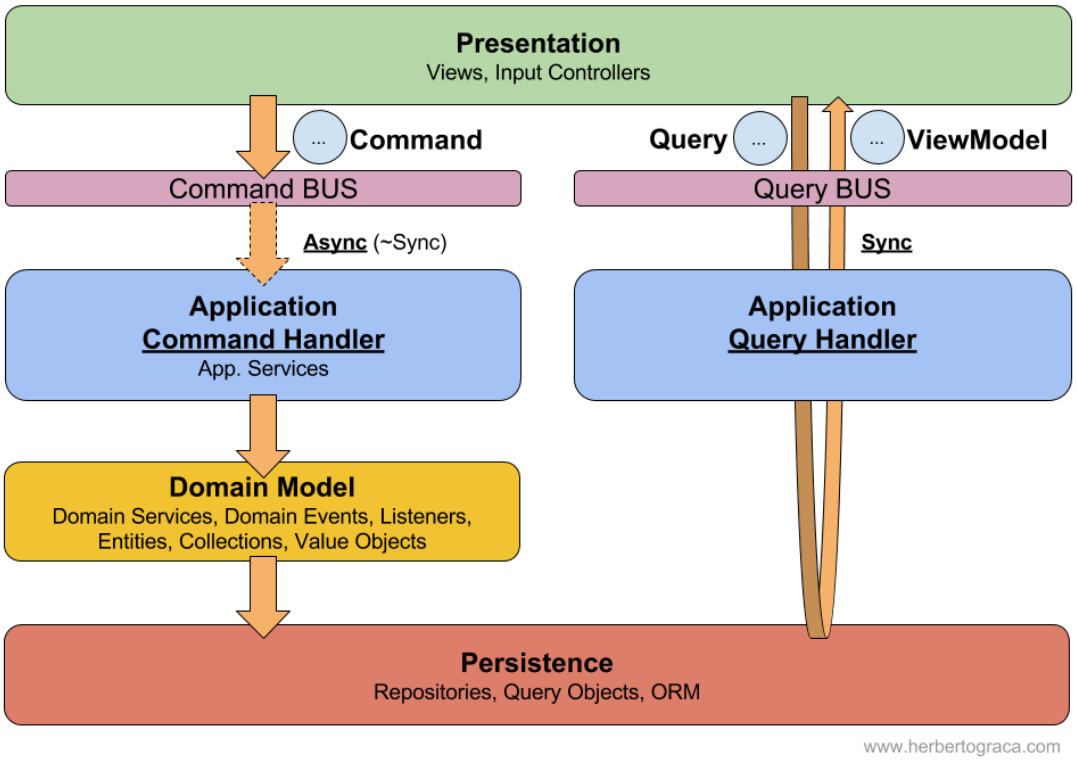
The usual way that the command bus achieves this is by using Decorators that wrap around the command bus (or a decorator already decorating it), in a matryoshka (https://carlalexander.ca/wp-content/uploads/2014/12/2318060944_bded6ecaf2_o-1038x576.jpg) kind of construction.



This allows us to create our own decorators and to configure the (maybe 3rd party) command bus to be composed by whatever decorators, in whatever order, adding our custom functionality to the command bus. If we need command queueing, we add a decorator to manage the queueing of commands. If we don't use a transactional DB, we don't need a decorator to wrap the handler execution around a DB transaction, and so forth, so on.

Command Query Responsibility Segregation

By putting together the concepts of CQS, Command, Query and CommandBus we finally reach CQRS. CQRS can be implemented in different ways and up to different levels, maybe only the Command side, or maybe not using a Command Bus. For the sake of completion, this is a diagram that represents how I see a full CQRS implementation:



Query side

Following CQS, the Query side will only return data, not changing it at all. Since we don't intend to execute a business process on that data, we don't need business objects (ie. entities), so we don't need an ORM to hydrate entities for us, and we don't need to get all the data needed to hydrate an entity. We just need to query for raw data to show to the user and exactly the data that we need in the template shown to the user!

This is a performance gain right here: When querying for data we don't need to go through the business logic layers to get it, we only do and get exactly what we need.

Because of this separation, another possible optimisation is to completely separate the data storage into two separated data storages: one optimised for writes and another one optimised for reads. For example, if we are using an RDBMS:

- The reads don't need any data integrity validation, they don't need foreign key constraints at all because data integrity validation is done when writing into the data storage. So **we can remove data integrity constraints from the read database**.
- We can also use **DB views with exactly the data that we need in each template**, making the querying trivial and therefore faster (although we will need to keep the view in sync with the template changes, adding to the system complexity).

At this point, if we have a specialised DB view for each template, which makes the querying trivial, why do we need an RDBMS for our reads?! Maybe we can **use a document storage for our reads**, like Mongo DB or even Redis, which are faster. Maybe yes, maybe not, I'm just saying it might be worth thinking about it if the application is having performance issues on the read side.

The querying itself can be done using a Query object that returns an array of data to be used in the template, or we can use something more sophisticated like a Query Bus that, for example, receives a template name, uses a query object to query the data and returns an instance of the ViewModel that

the template needs.

This approach can solve several problems identified by Greg Young (https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf):

- *Large numbers of read methods on repositories often also including paging or sorting information;*
- *Getters exposing the internal state of domain objects in order to build DTOs;*
- *Use of prefetch paths on the read use cases as they require more data to be loaded by the ORM;*
- *Loading of multiple aggregate roots to build a DTO causes non-optimal querying to the data model. Alternatively, aggregate boundaries can be confused because of the DTO building operations;*
- *The biggest problem though, is that the optimisation of queries is extremely difficult: Because queries are operating on an object model and then being translated into a data model, likely by an ORM, it can be very difficult to optimise these queries.*

Command side

As explained earlier, using commands we shift the application from a data-centric design to a behavioural design, which comes in line with Domain Driven Design.

By removing the read operations from the code that processes commands, from the Domain, the problems identified by Greg Young just vanish:

- *Domain objects suddenly no longer have a need to expose internal state;*
- *Repositories have very few if any query methods aside from GetById;*
- *A more behavioural focus can be had on Aggregate boundaries.*

The "one-to-many" and "many-to-many" relationships between entities can have a severe impact on the ORM performance. The good news is that we seldom need those relations when processing commands, they are mostly used for querying and we have just moved the querying away from the command processing so we can remove those entity relationships. I'm not talking here about the relations between tables in an RDBMS, those foreign key constraints should still exist in the write DB, I'm talking about the connections between entities which are configured at the ORM level.

> *Do we really need a collection of orders on the customer entity? In what command would we need to navigate that collection? In fact, what kind of command would need any one-to-many relationship? And if that's the case for one-to-many, many-to-many would definitely be out as well. I mean, most commands only contain one or two IDs in them anyway.*
>
> *Udi Dahan 2009, Clarified CQRS (http://udidahan.com/2009/12/09/clarified-cqrs/)*

Following the same thoughts as for the Query side, if the write side is not being used for complex queries, could we replace the RDBMS for a document or key-value storage with the serialised entities? Maybe yes, maybe not, I'm just saying it might be worth thinking about it if the application is having performance issues on the write side.

Business process events

After a command is processed, and if it was successfully processed, the handler triggers an event notifying the rest of the application about what has happened. The events should be named as the command that triggered it, except that, as is the rule with events, it should be in the past tense.

Conclusion

By using CQRS we can completely separate the read model from the write model, allowing us to have optimised read and write operations. This adds to performance, but also to the clarity and simplicity of the codebase, to the ability of the codebase to reflect the domain, and to the maintainability of the codebase.

Again, it's all about encapsulation, low coupling, high cohesion, and the Single Responsibility Principle.

Nevertheless, it's good to keep in mind that although CQRS provides for a design style and several technical solutions that can make an application very robust, that doesn't mean that all applications should be built this way: We should use what we need, when we need it.

Sources

(The ones in **bold** are the ones I consider most valuable.)

1994 – Gamma, Helm, Johnson, Vlissides – Design Patterns: Elements of Reusable Object-Oriented Software (https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented-ebook/dp/B000SEIBB8)

1999 – Bala Paranj – Java Tip 68: Learn how to implement the Command pattern in Java (http://www.javaworld.com/article/2077569/core-java/java-tip-68--learn-how-to-implement-the-command-pattern-in-java.html)

2004 – Eric Freeman, Elisabeth Robson – Head First Design Patterns (https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124)

2005 – Martin Fowler – Command Query Separation (https://martinfowler.com/bliki/CommandQuerySeparation.html)

2009 – Udi Dahan – Clarified CQRS (http://udidahan.com/2009/12/09/clarified-cqrs/)

2010 – Greg Young – CQRS, Task Based UIs, Event Sourcing agh! (http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/)

2010 – Greg Young – CQRS Documents (https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf)

2010 – Udi Dahan – Race Conditions Don't Exist (http://udidahan.com/2010/08/31/race-conditions-dont-exist/)

2011 – Martin Fowler – CQRS (https://martinfowler.com/bliki/CQRS.html)

2011 – Udi Dahan – When to avoid CQRS (http://udidahan.com/2011/04/22/when-to-avoid-cqrs/)

2014 – Greg Young – CQRS and Event Sourcing – Code on the Beach 2014 (https://www.youtube.com/watch?v=JHGkaShoyNs)

2015 – Matthias Noback – Responsibilities of the command bus (https://php-and-symfony.matthiasnoback.nl/2015/01/responsibilities-of-the-command-bus/)

2017 – Martin Fowler – What do you mean by "Event-Driven"? (https://martinfowler.com/articles/201701-event-driven.html)

2017* – Doug Gale – Command Pattern (http://wiki.c2.com/?CommandPattern)

2017* – Wikipedia – Command Pattern (https://en.wikipedia.org/wiki/Command_pattern)

* Viewed in

Tagged:
CQRS,
Event Sourcing,
Greg Young,
software architecture,
software developement,
Udi Dahan

Published by hgraca

View all posts by hgraca

9 thoughts on "From CQS to CQRS"

Chris Cooper says:
August 3, 2019 at 00:52
"Nevertheless, there are some patterns that are exceptions to this rule as, again, Martin Fowler says, the traditional queue and stack implementations pop an element of the queue or stack, both changing the queue/stack and returning the element removed from it."

Meyer doesn't implement stacks/queues this way – there is no "pop" operation, only an "item" feature to return an element, and a "remove" operation that does not return an element.

↳ Reply
Rey says:
May 24, 2018 at 17:38
Hello, actually I have more clear what CQRS is but where can I see and example to figure out. Best Regards

↳ Reply

hgraca says:
May 24, 2018 at 17:42
Im working on some code sample of how it can be done, although there are as many ways to do it as there are developers. I should be finished with it this quarter.

↳ Reply

Alan Gabriel Bem says:
December 10, 2018 at 11:00
Yup, you are right. For example in my interpretation I am heavily using command handler decorators alongside command bus decorators.
Also I've seen implementation utilising middleware/plugin systems instead of decoration – Don't like this approach personally.

↳ Reply

Pingback: CQRS in 4 steps – presentation | Radek Maziarka Blog
Pingback: Command Query Responsibility Segregation (CQRS) – Java Việt Nam
Pingback: The Software Architecture Chronicles – @herbertograca

Jeroen Nas says:
October 20, 2017 at 23:29
Very nice overview. Nice to see that we are going in similar directions.

↳ Reply

Timothy Verdonck says:
October 19, 2017 at 21:14
Thanks for this article!

↳ Reply