let's **move**
the **java world**

CQRS for Great Good

Oliver Wolf

geecon 2013

Oliver Wolf
@owolf

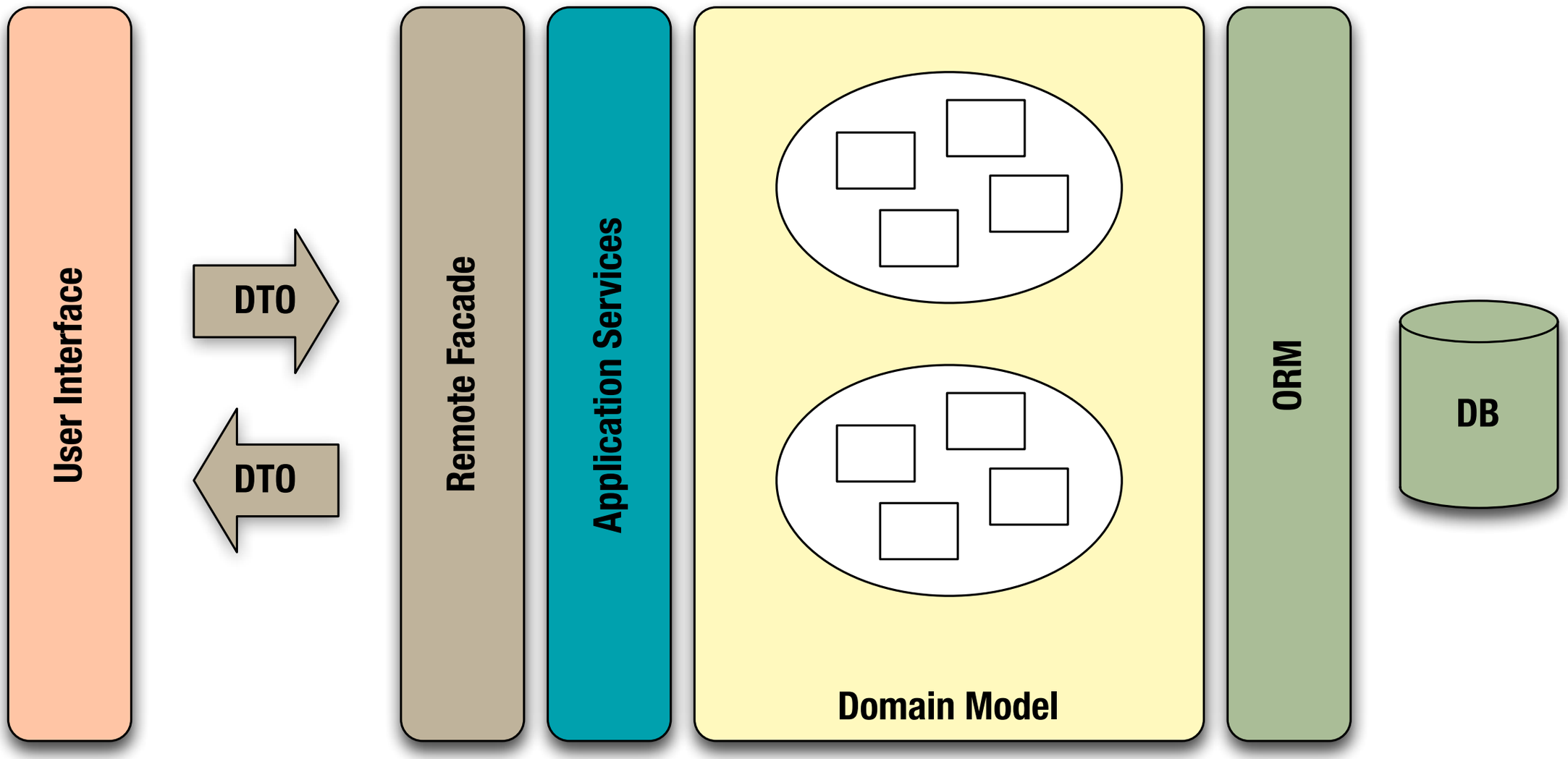
www.innoQ.com
@innoQ

CQRS

CQRS

Command Query Responsibility Segregation

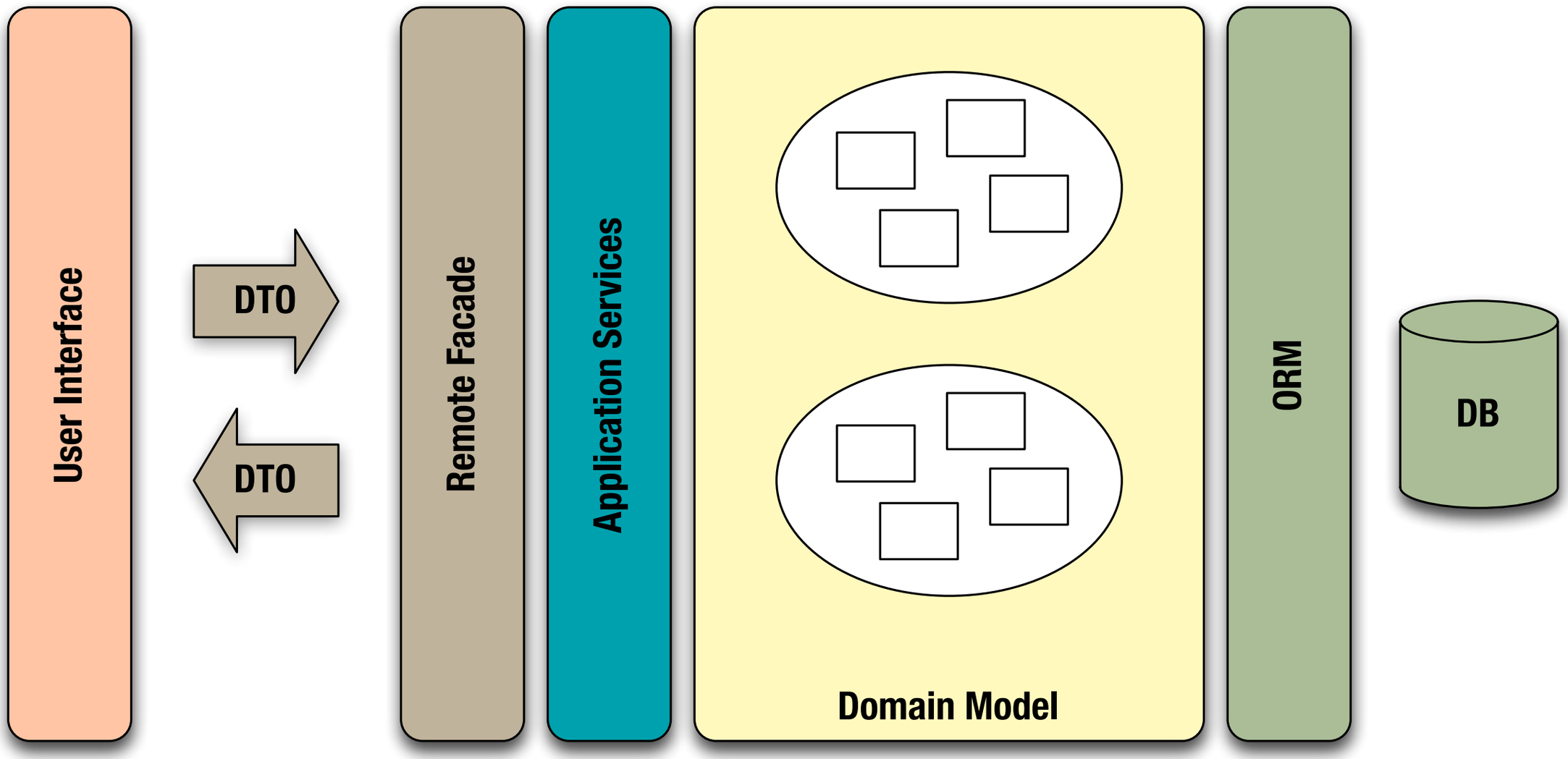
The default architecture for distributed business apps



The default architecture for distributed business apps

```
<customer>
    <name>John Doe</name>
    <address>...</address>
    ...
</customer>
```

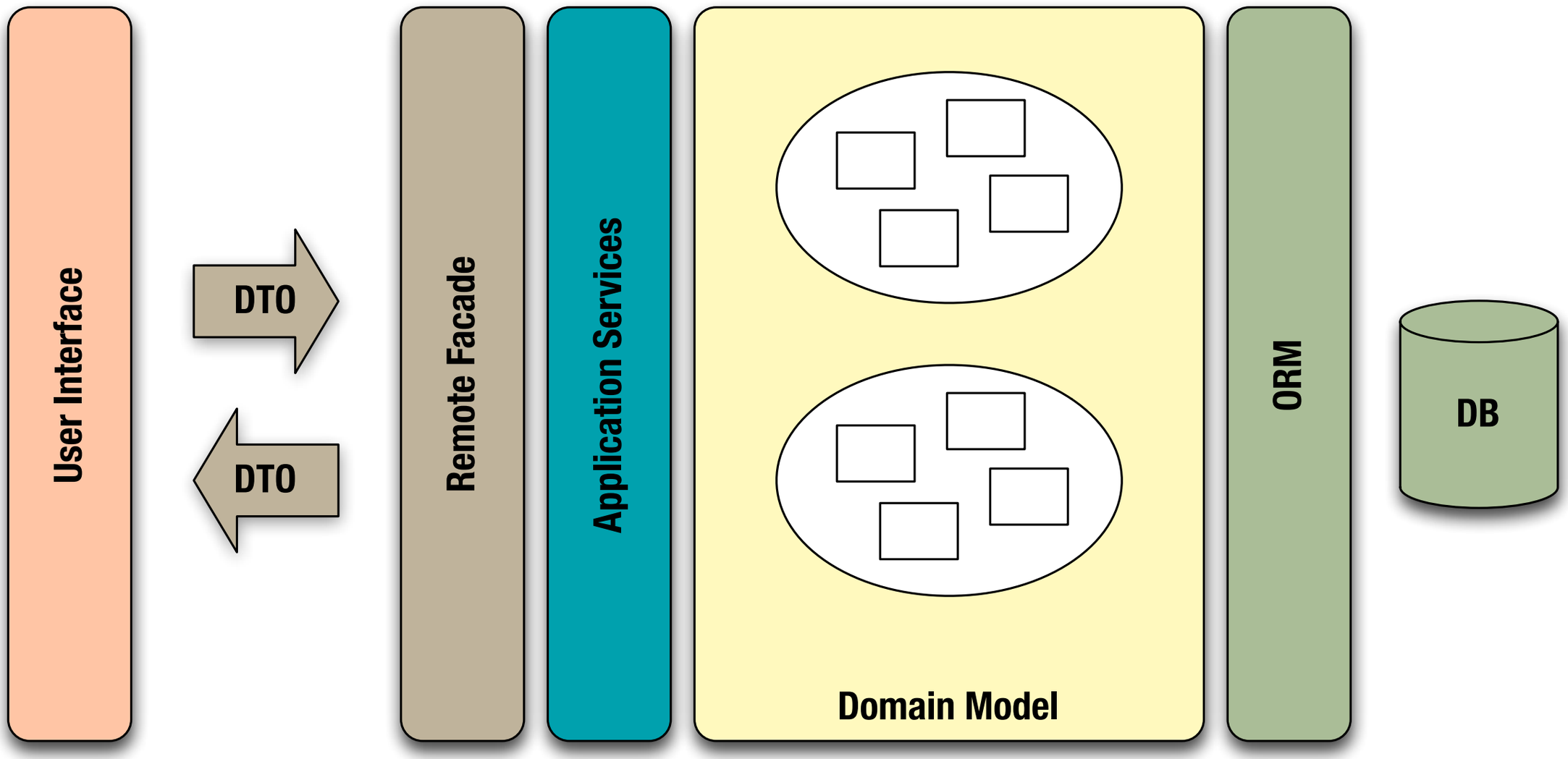
```
findCustomers()
getCustomer()
updateCustomer()
```



The default architecture for distributed business apps

```
{
   "name": “John Doe”,
   "address": {
       ...
   }
}
```

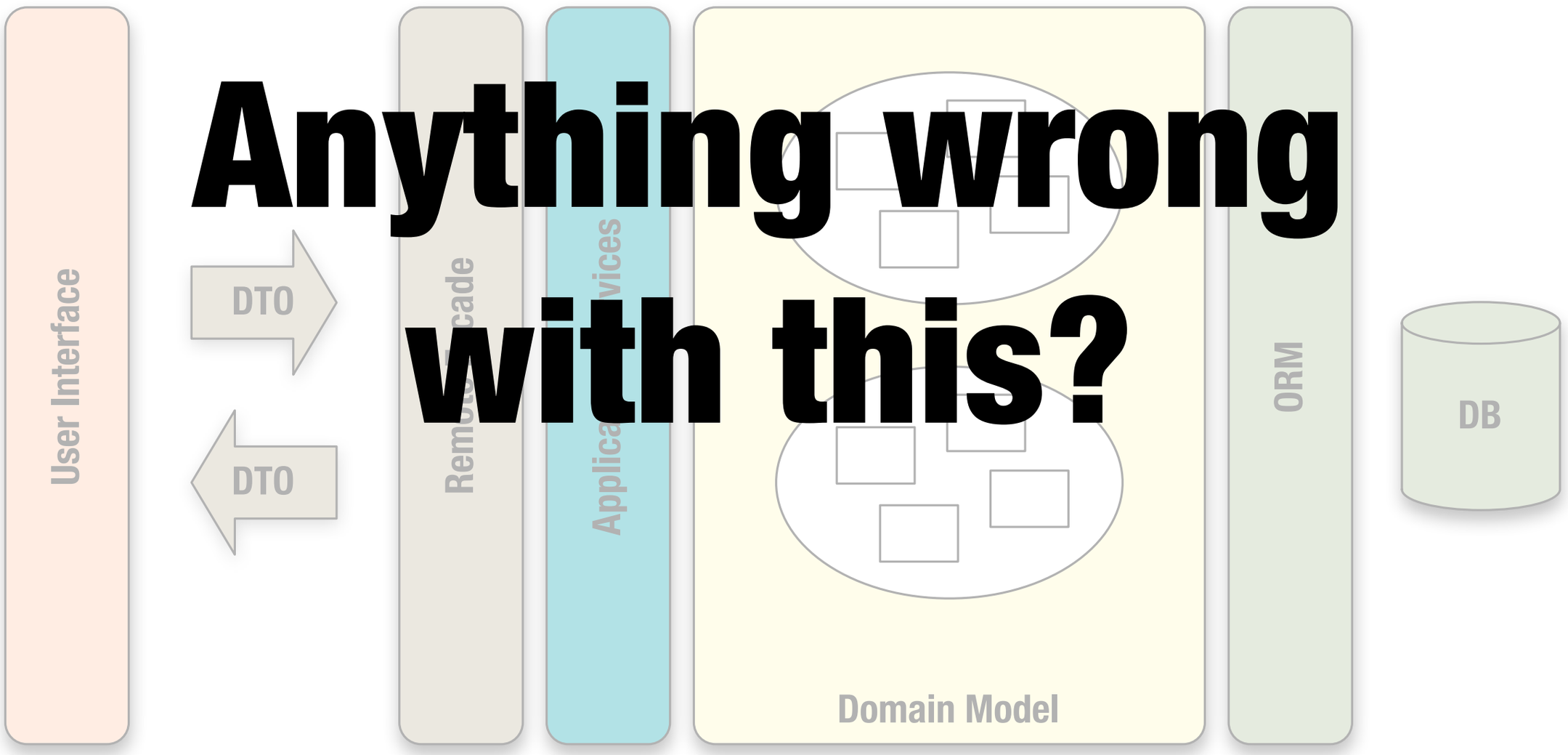
```
GET /customers?filter=...
GET /customer/{id}
PUT /customer/{id}
```



The default architecture for distributed business apps

```
{
   "name": “John Doe”,
   "address": {
       ...
   }
}
```

```
GET /customers?filter=...
GET /customer/{id}
PUT /customer/{id}
```



Anything wrong with this?

Maybe not.

Scalability?

Domain Model?


Text

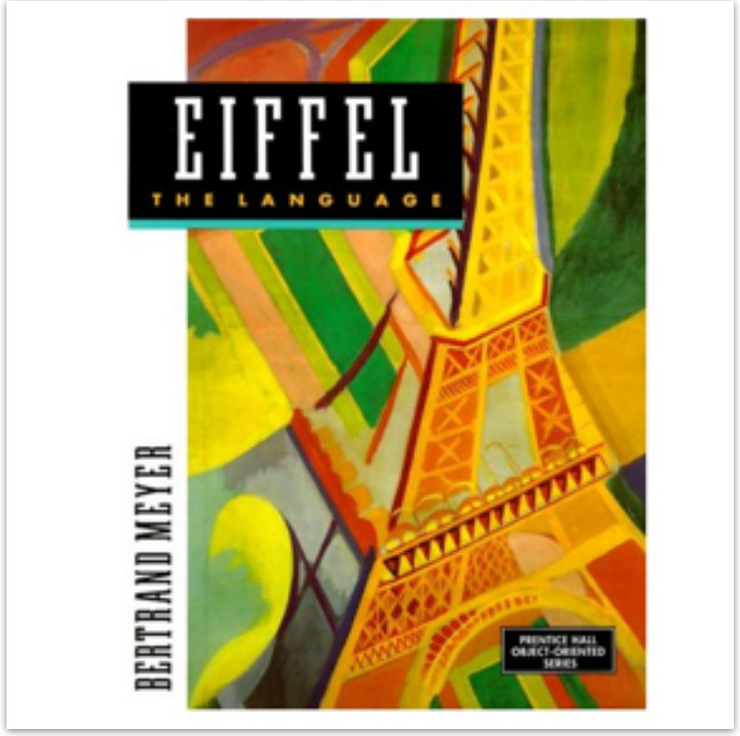

Text
Bertrand Meyer
ETH Zurich


“The features that characterize a class are divided into commands and queries. A command serves to modify objects, a query to return information about objects.”
Text
Bertrand Meyer
ETH Zurich
cited from: Object-Oriented Software Construction, second edition, 1997

CQS

Command Query Separation

Part of the Design-by-Contract methodology



First demonstrated in the object-oriented EIFFEL programming language

```
class Foo {
    void command();
    Result query();
}
```

```
class Foo {
   void command();
   Result query();
}
```

Mutates state

```
class Foo {
    void command();
    Result query();
}
```

Mutates state

Returns a value without causing side effects

CQRS

–

CQS in the large

CQRS

–

CQS in the large

Scope is a single class

Scope is a Bounded Context

CQRS

–

CQS in the large

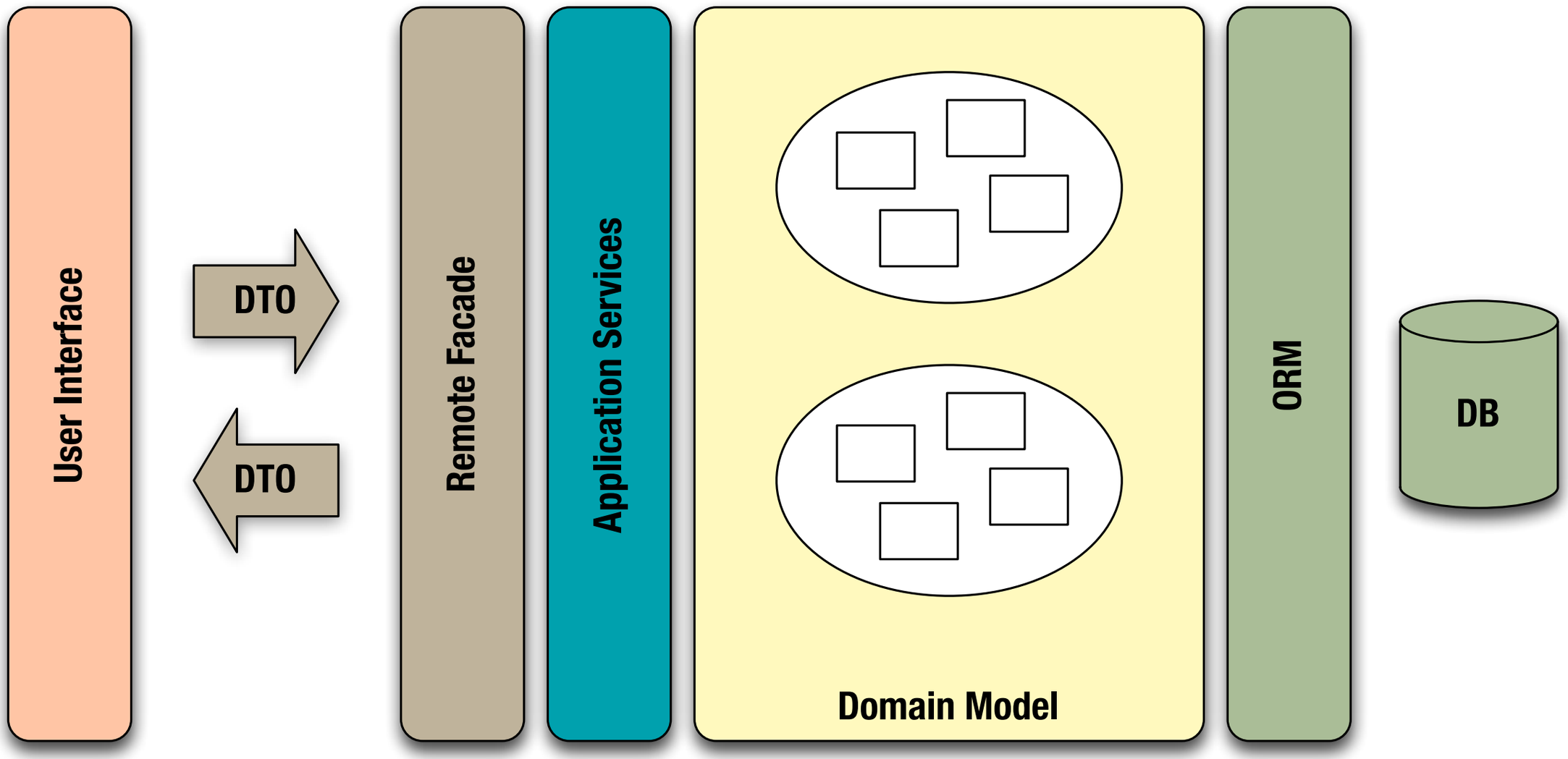
Scope is a single class

```
interface CustomerService {
  void updateCustomer(Customer);
  CustomerList findCustomers(CustomerQuery);
  Customer getCustomer(ID);
  void deleteCustomer(ID);
}
```

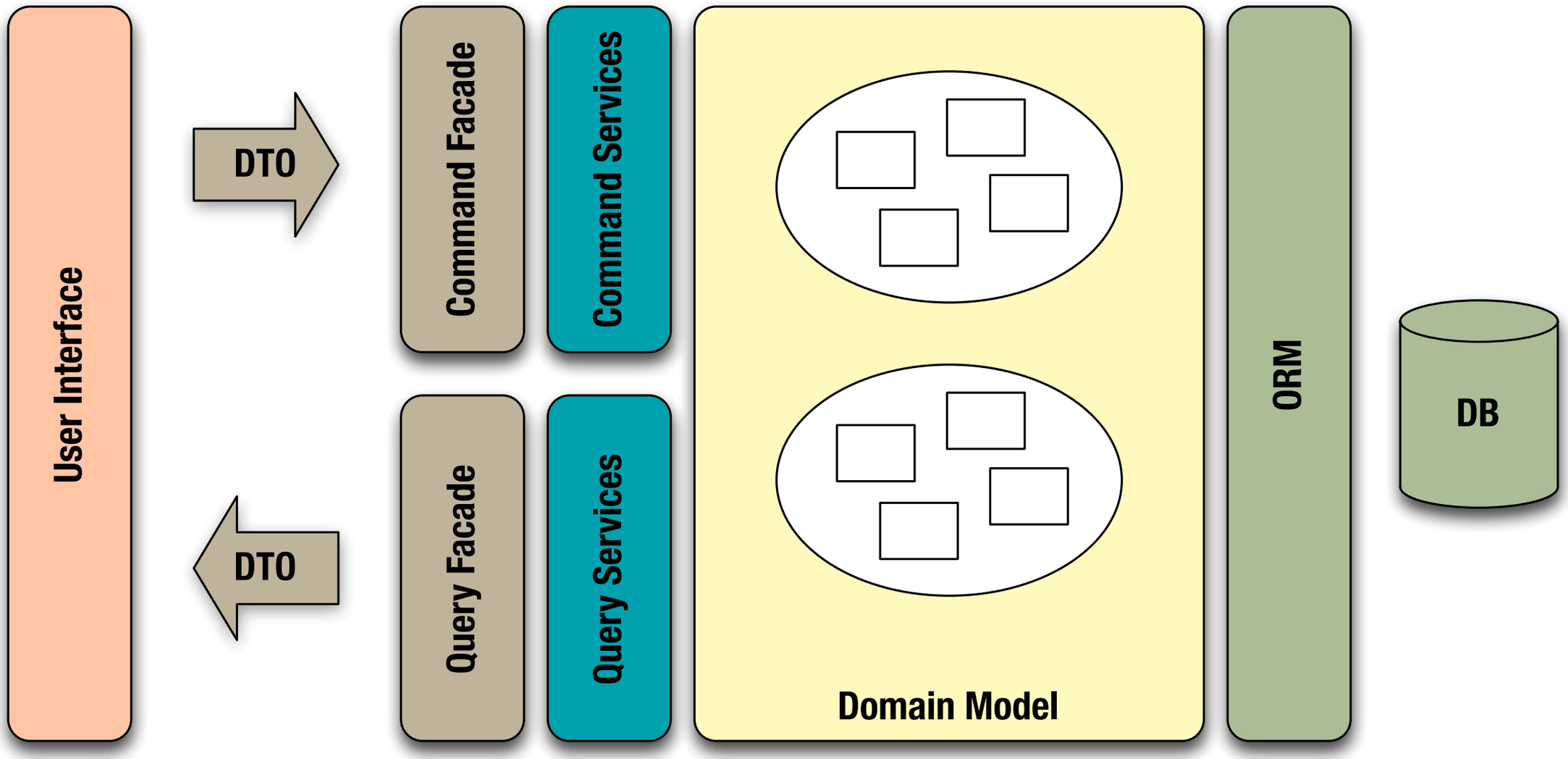
```
interface CustomerQueryService {
  CustomerList findCustomers(CustomerQuery);
  Customer getCustomer(ID);
}
```

```
interface CustomerCommandService {
  void updateCustomer(Customer);
  void deleteCustomer(ID);
}
```

The default architecture for distributed business apps

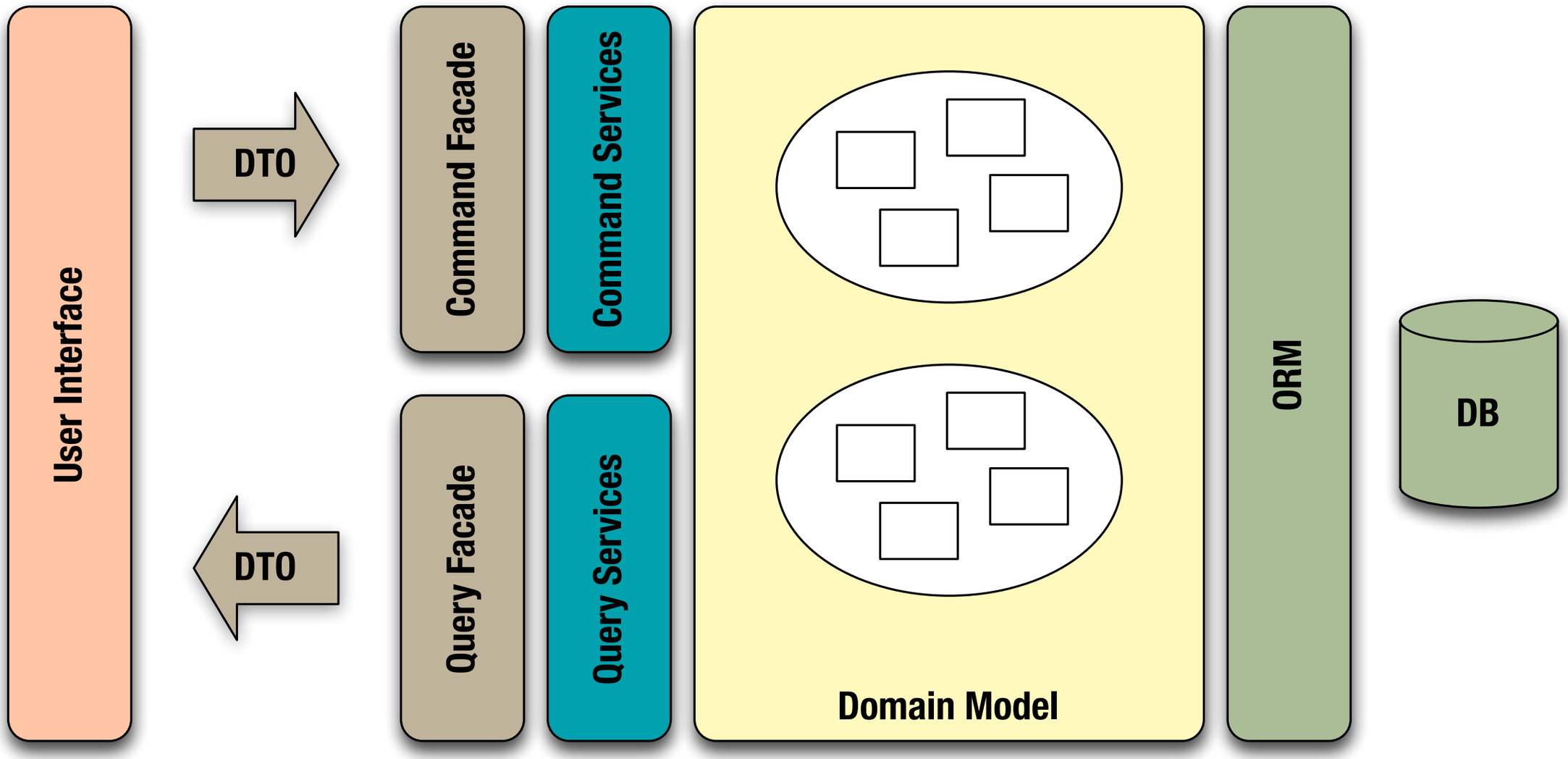


The default architecture for distributed business apps



The default architecture for distributed business apps

CQRS PATTERN APPLIED



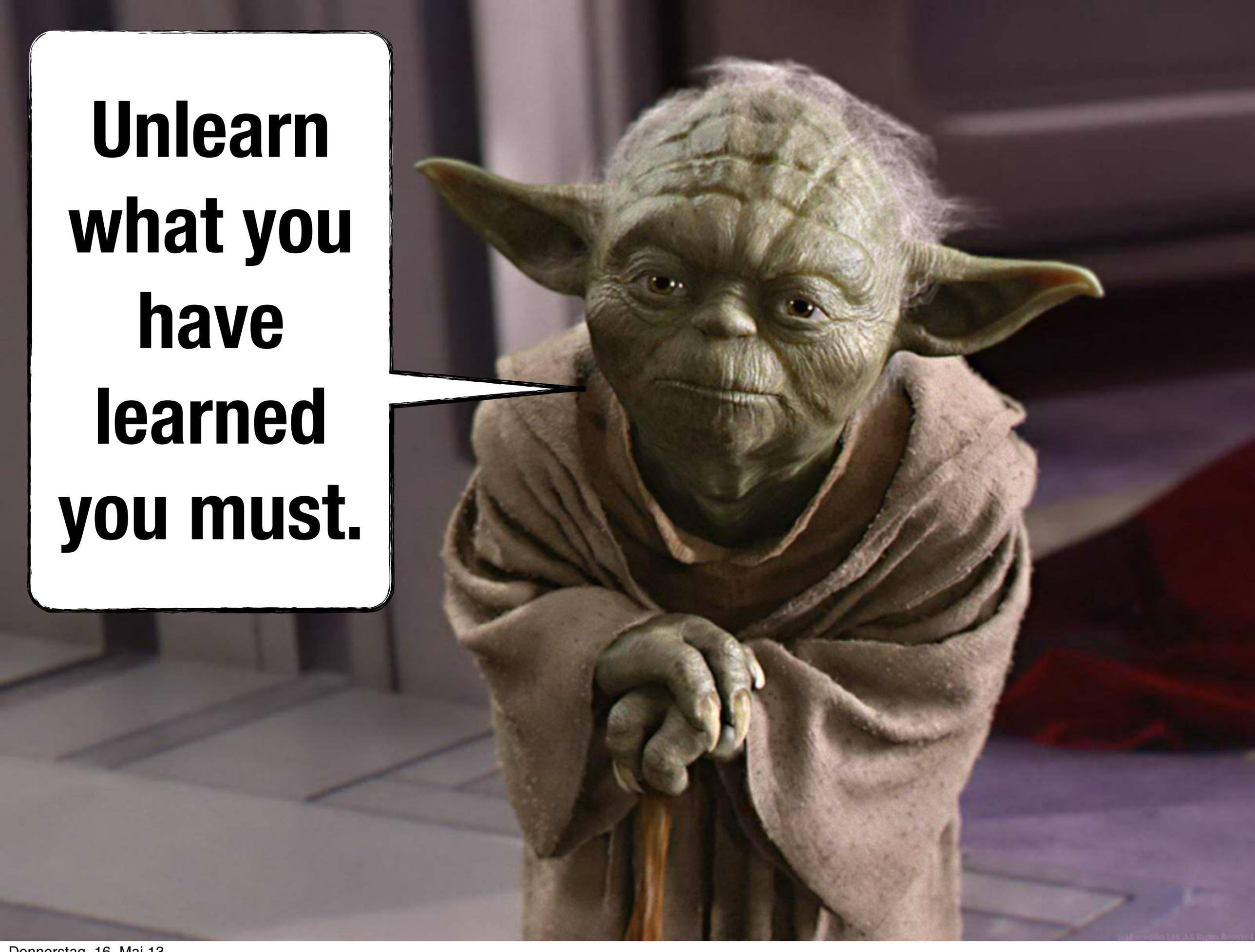

That’s it?
You
serious???

Yes, sorry.

The interesting thing about CQRS is not the pattern itself.

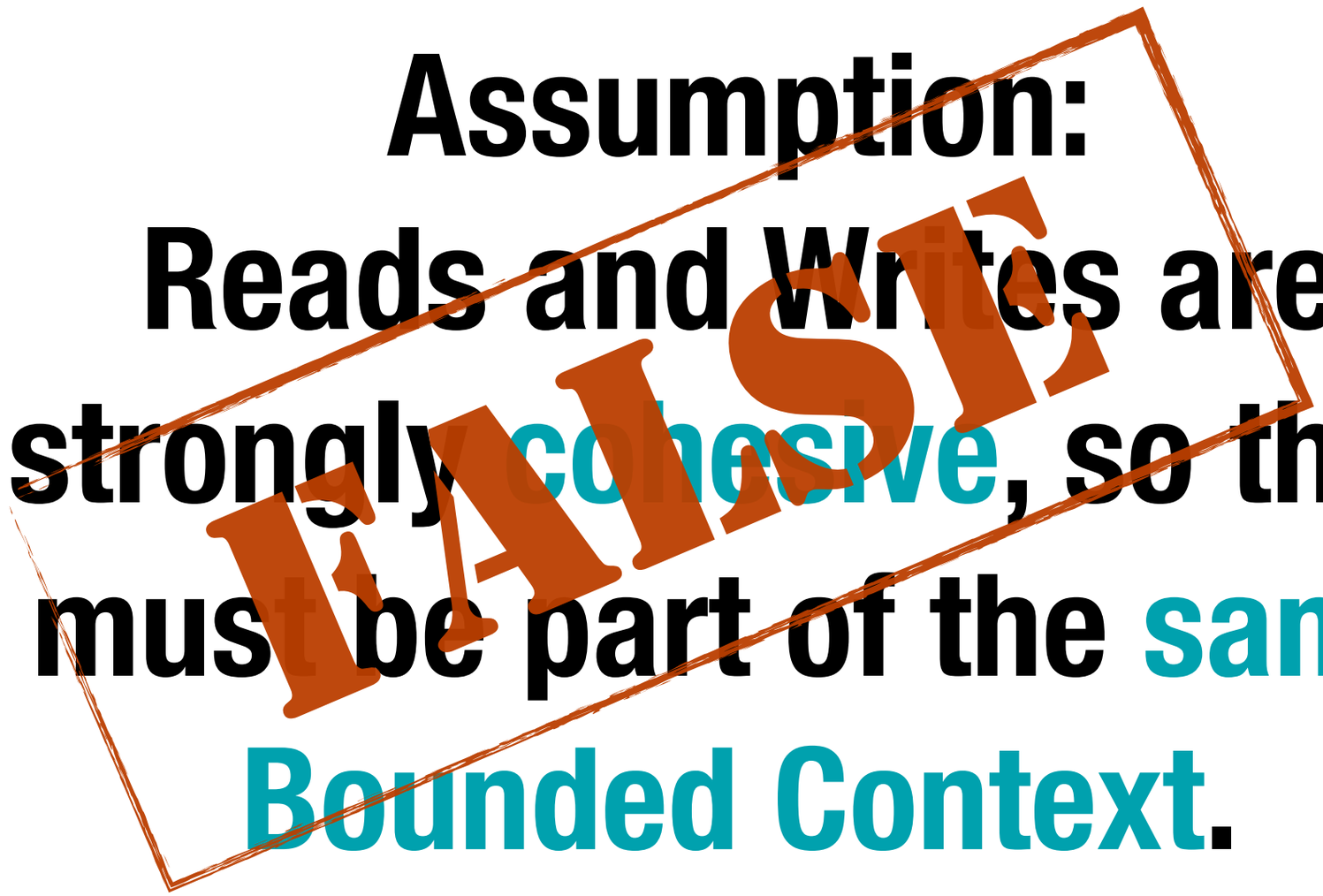
It’s damn simple, actually.

But it allows you to challenge established assumptions and opens up new architectural options!


Unlearn what you have learned you must.

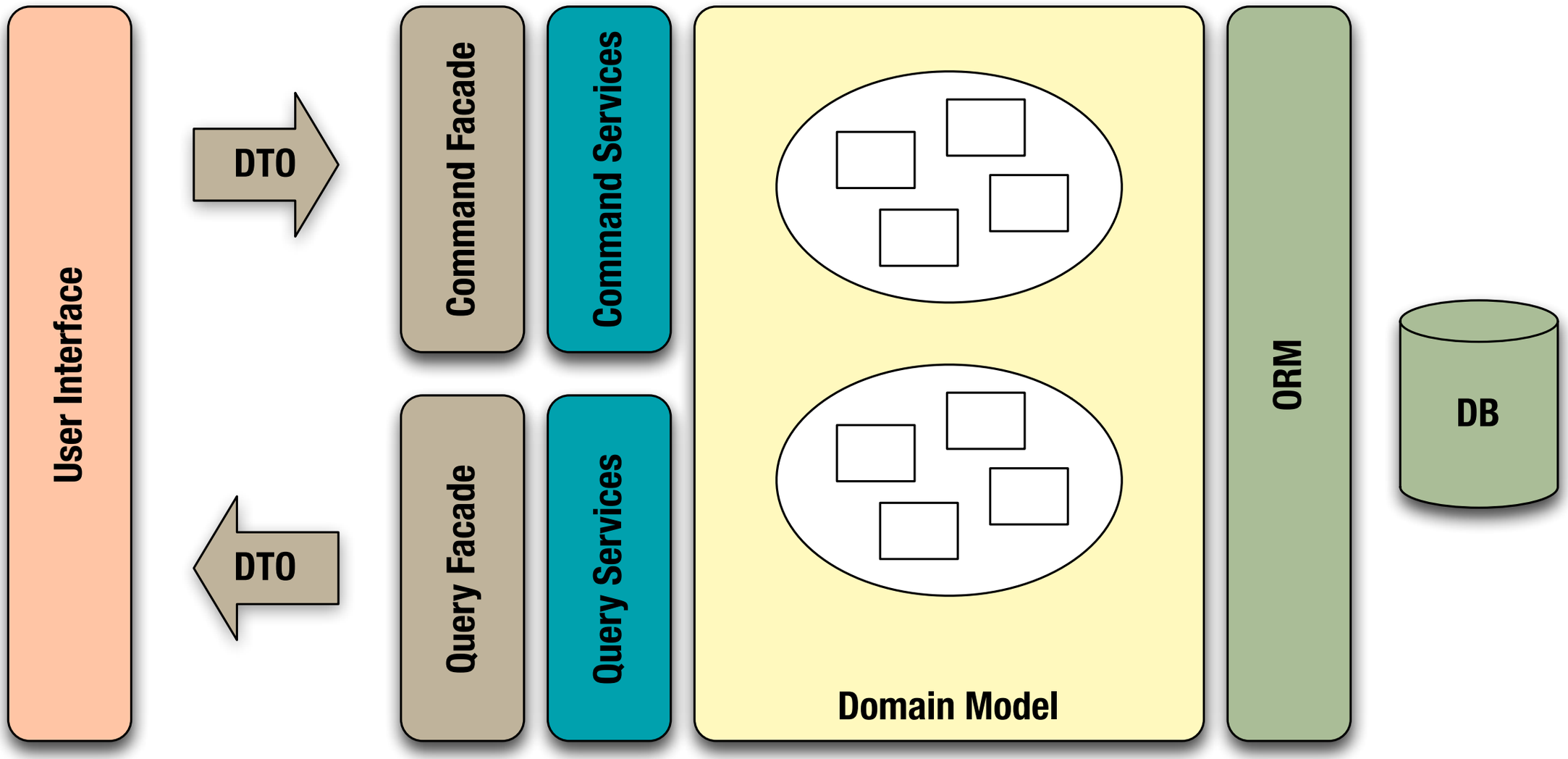
Assumption:
Reads and Writes are strongly cohesive, so they must be part of the same Bounded Context.

Assumption:
Reads and Writes are strongly cohesive, so they must be part of the same Bounded Context.



CQRSified

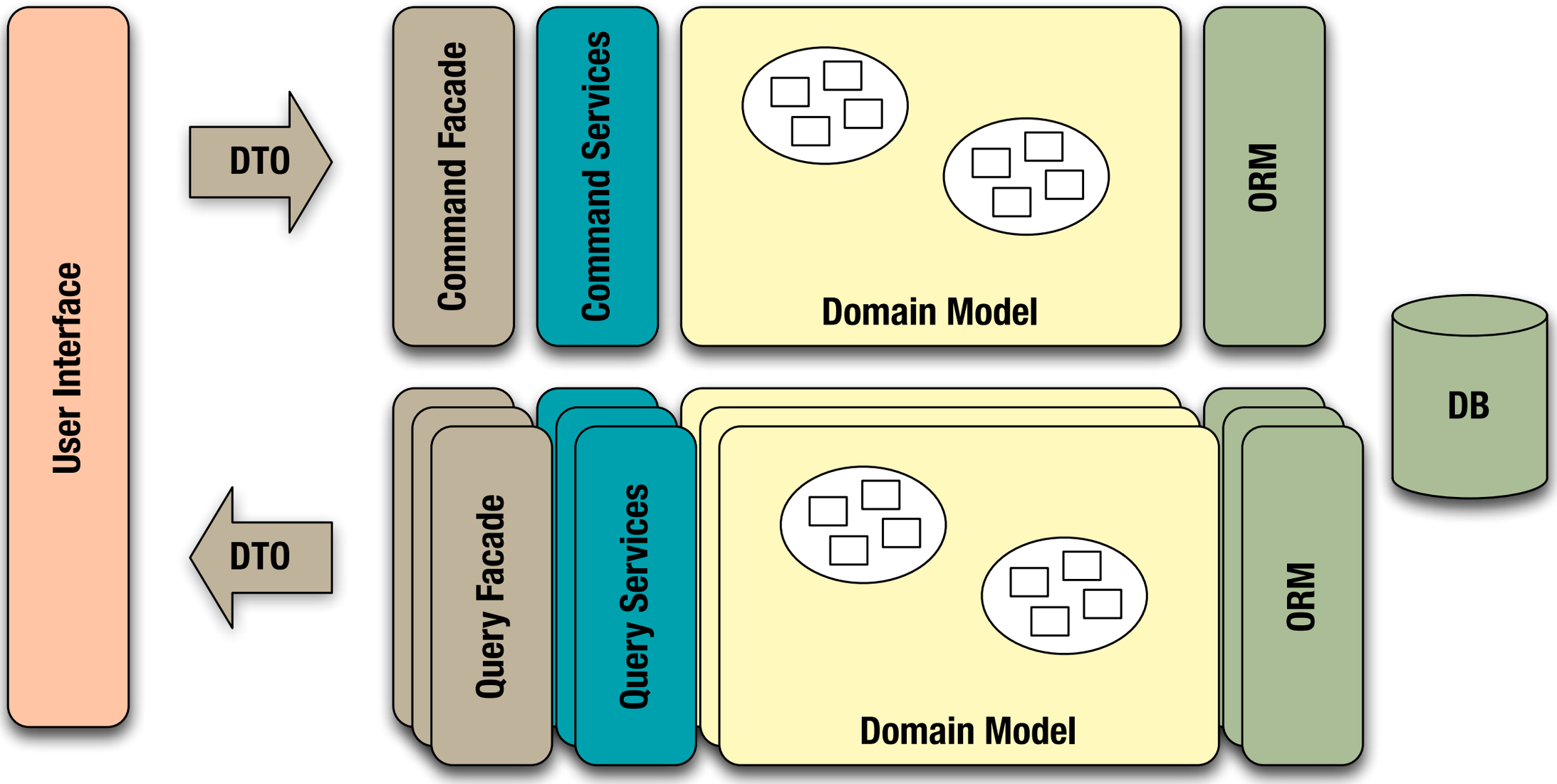
The ~~default~~ architecture for distributed business apps



CQRSified

The ~~default~~ architecture for distributed business apps

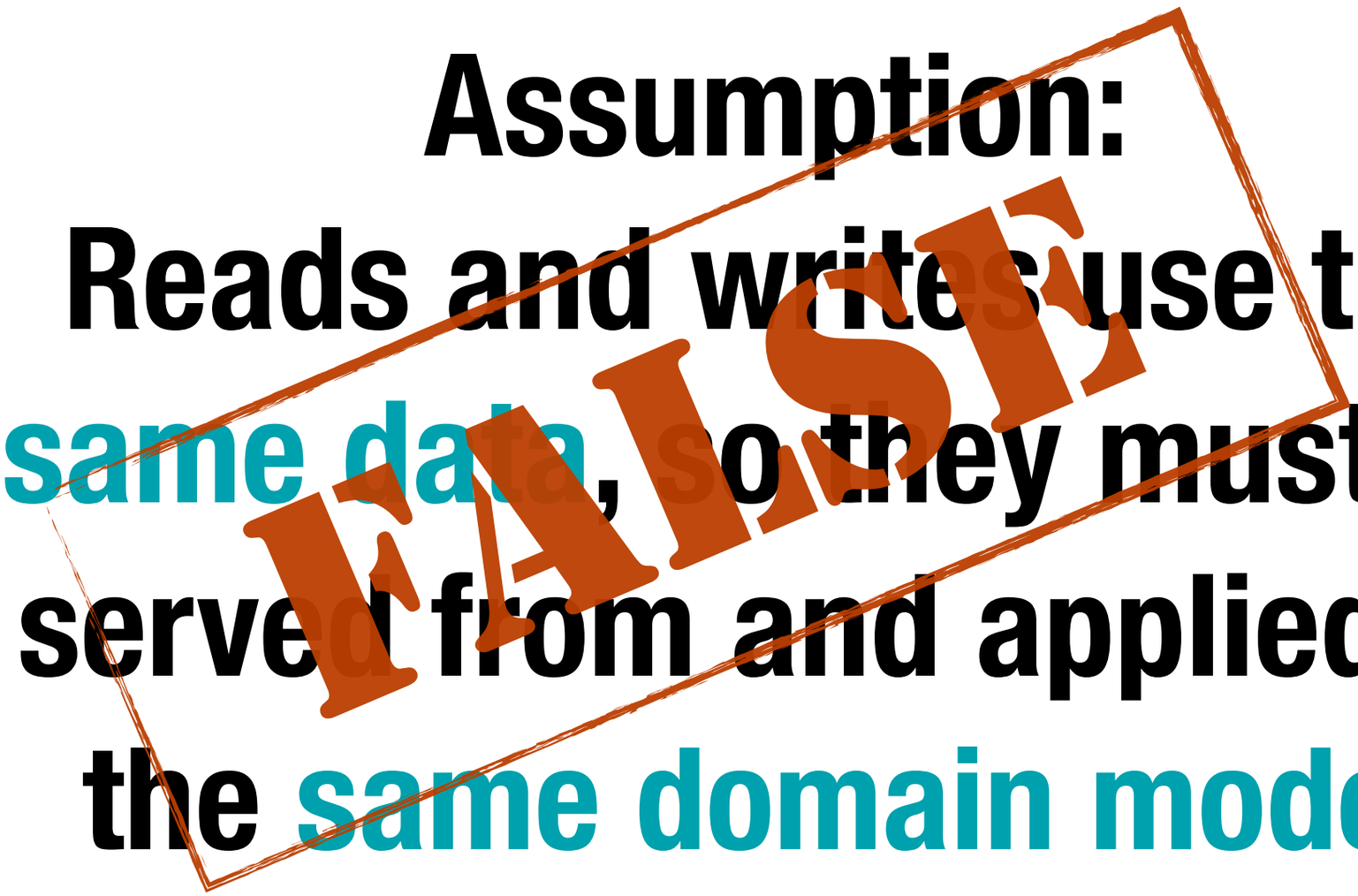
Command and query parts can scale independently,
e.g. to accommodate highly asymmetric load.



Assumption:
Reads and writes use the same data, so they must be served from and applied to the same domain model.

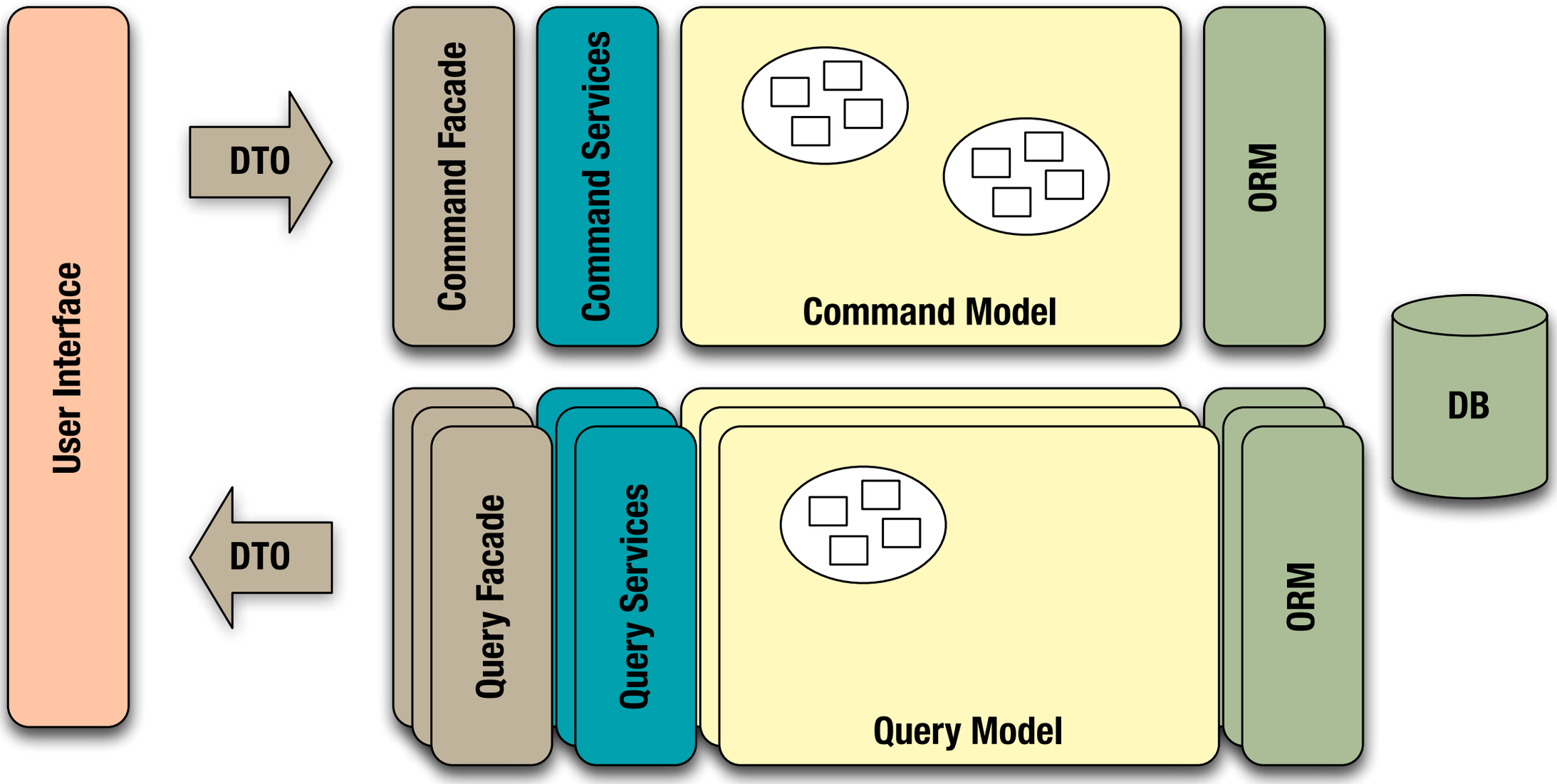
Assumption:

Reads and writes use the same data, so they must be served from and applied to the same domain model.



CQRSified

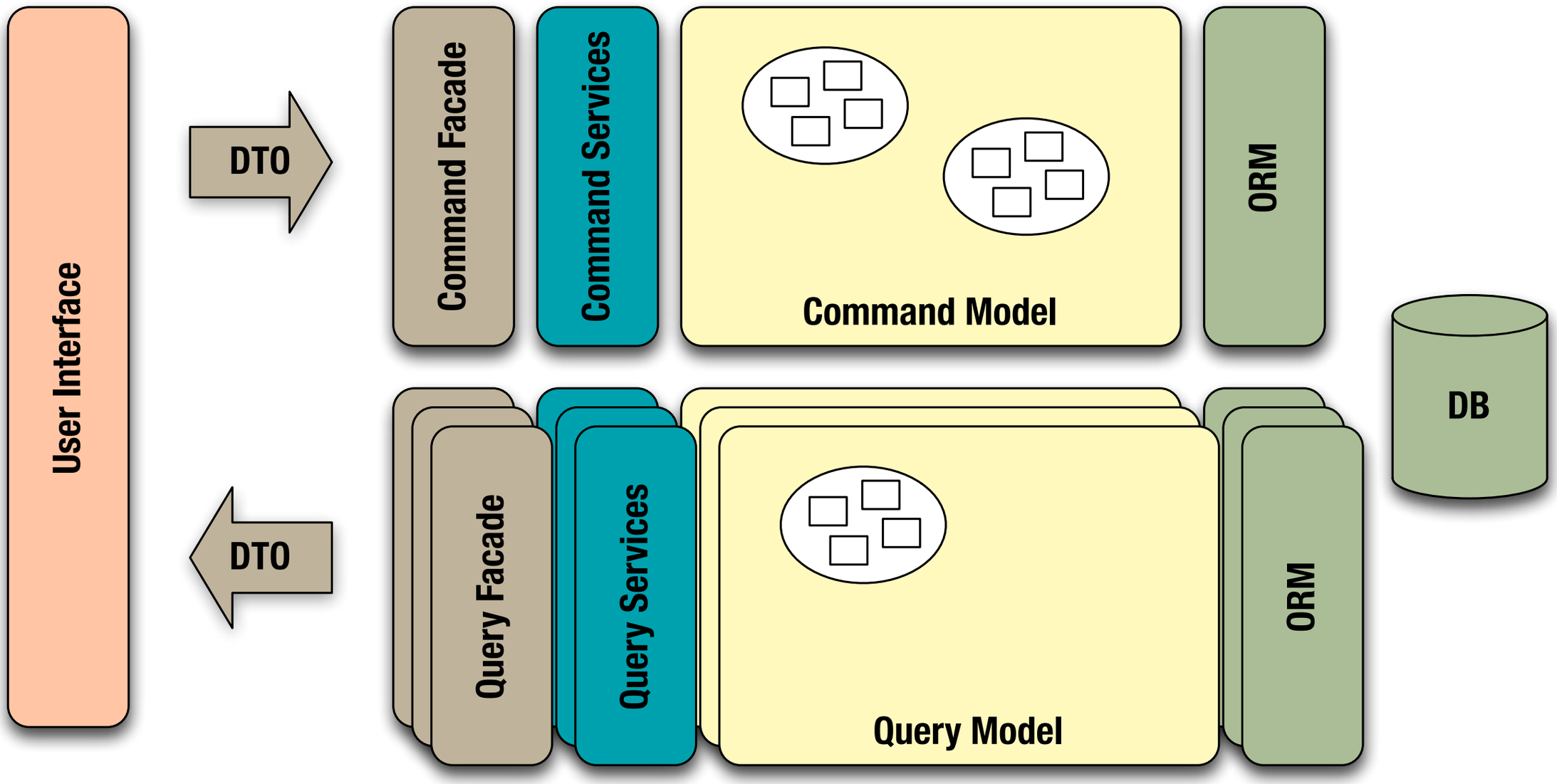
The ~~default~~ architecture for distributed business apps



CQRSified

The ~~default~~ architecture for distributed business apps

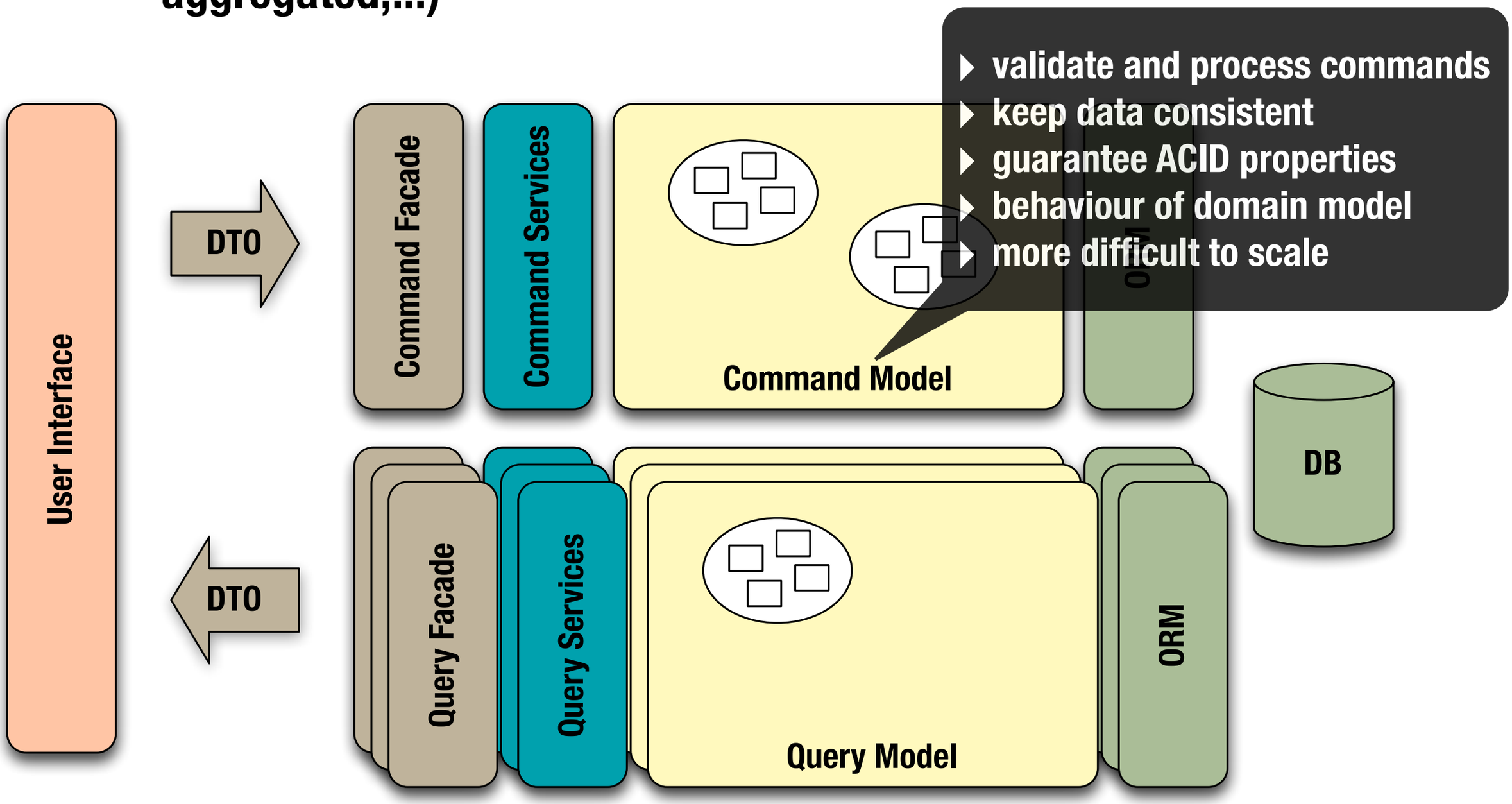
Queries can benefit from a specialized query model, optimized for quick data retrieval (de-normalized, pre-aggregated,...)



CQRSified

The ~~default~~ architecture for distributed business apps

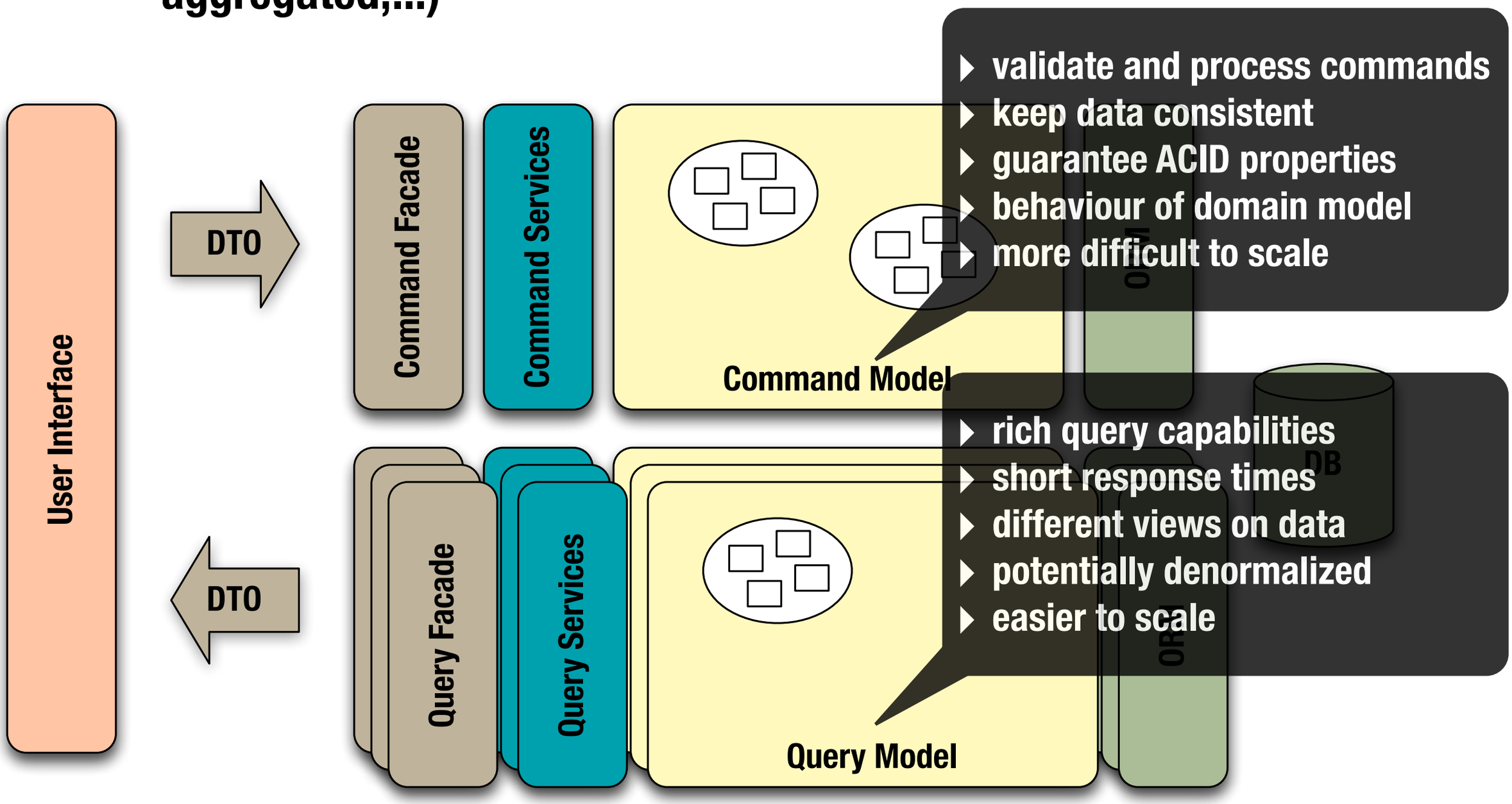
Queries can benefit from a specialized query model, optimized for quick data retrieval (de-normalized, pre-aggregated,...)



CQRSified

The ~~default~~ architecture for distributed business apps

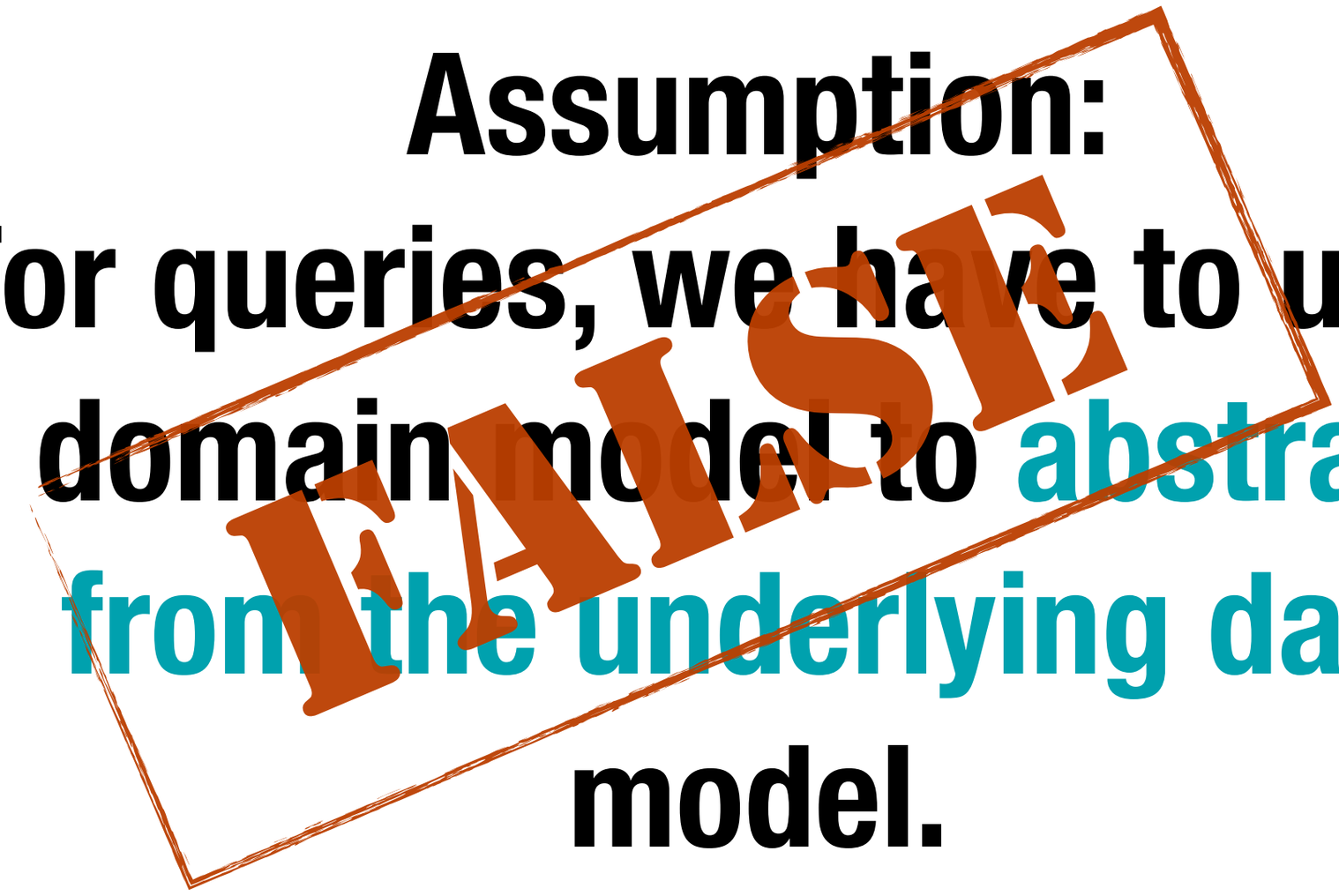
Queries can benefit from a specialized query model, optimized for quick data retrieval (de-normalized, pre-aggregated,...)



Assumption:
For queries, we have to use a domain model to abstract from the underlying data model.

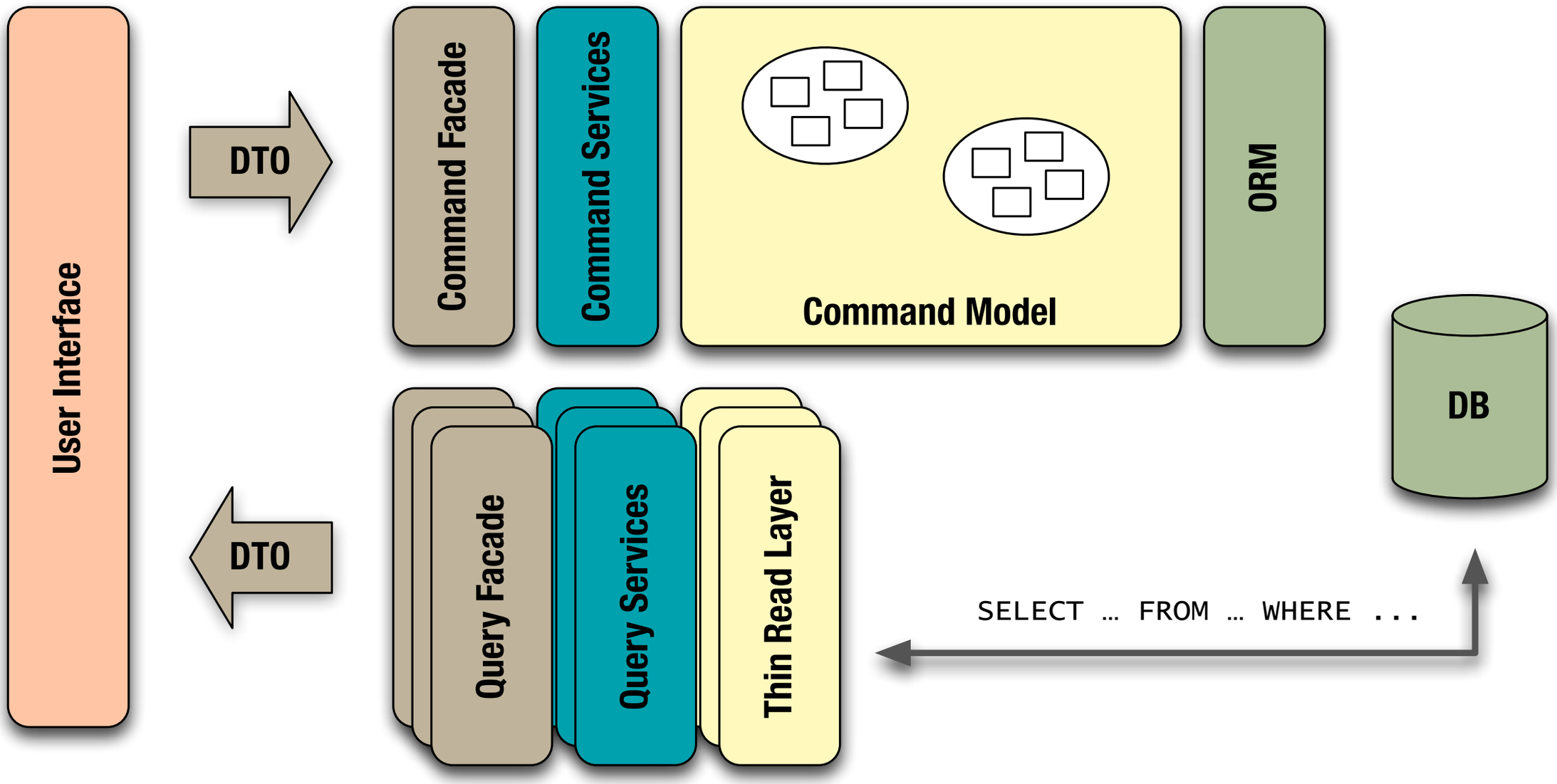
Assumption:

For queries, we have to use a domain model to abstract from the underlying data model.



CQRSified

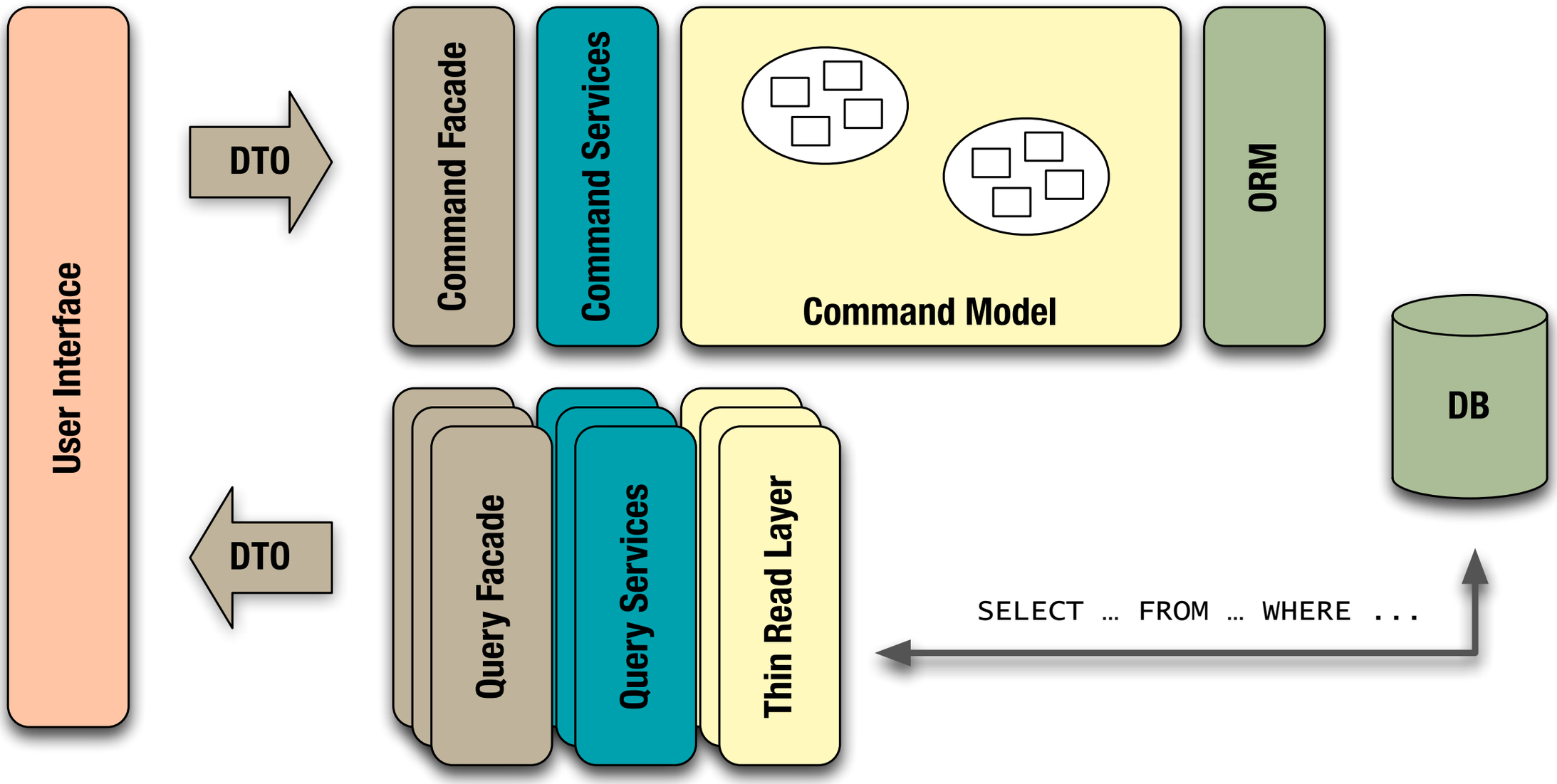
The ~~default~~ architecture for distributed business apps



CQRSified

The ~~default~~ architecture for distributed business apps

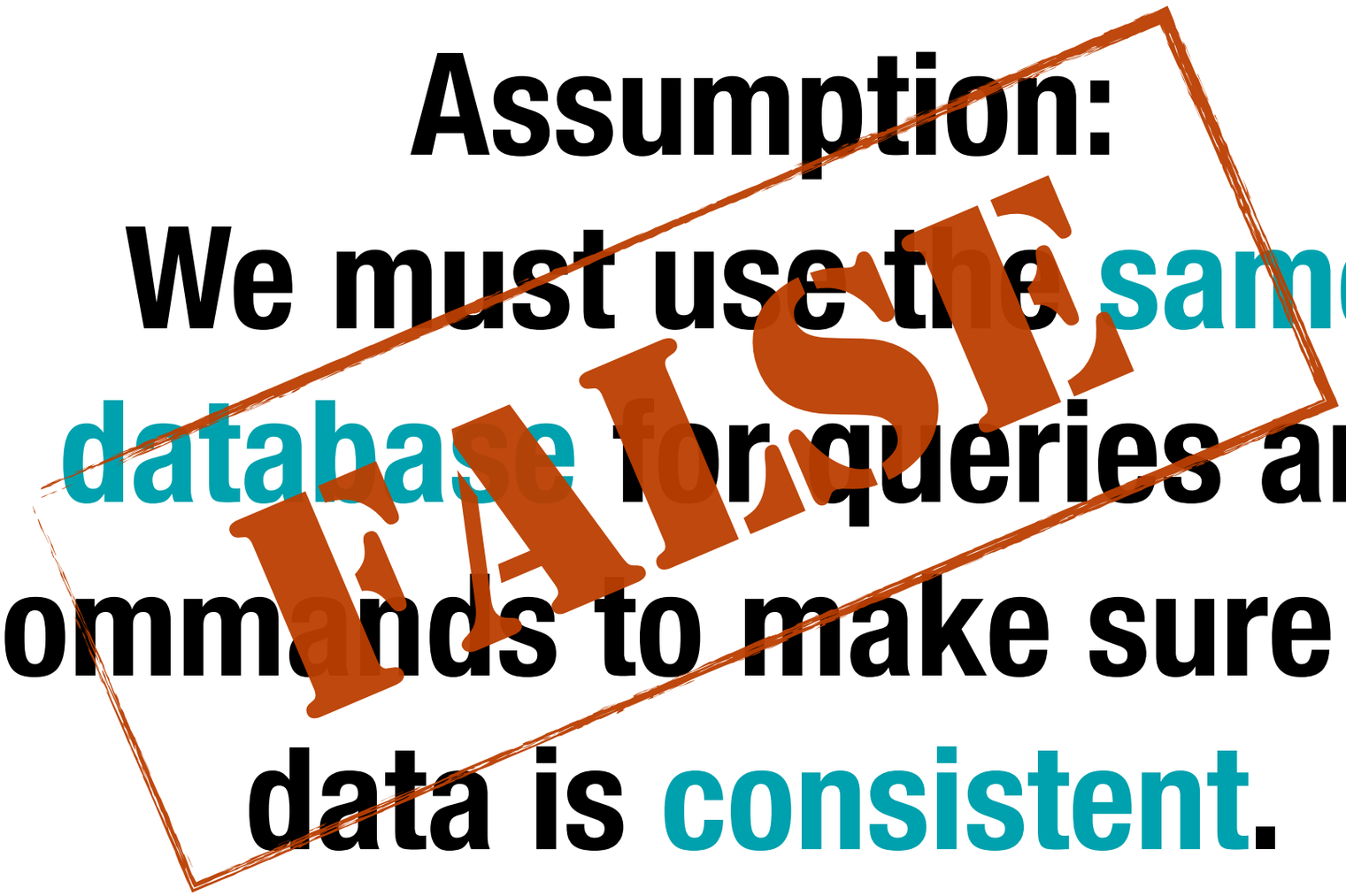
Queries are just dealing with data, not with behavior.
Why do we need objects?



Assumption:
We must use the same database for queries and commands to make sure that data is consistent.

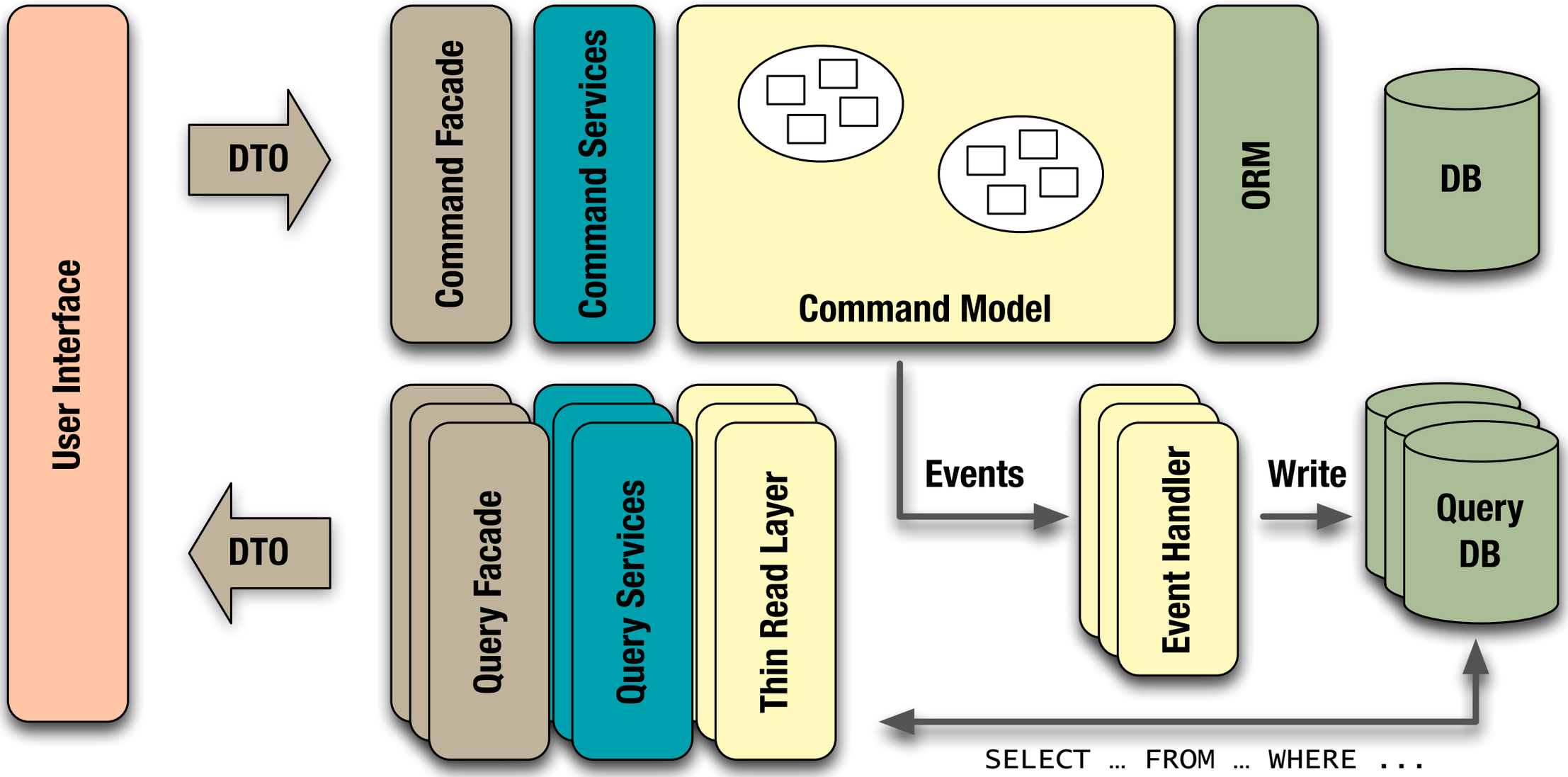
Assumption:

We must use the same database for queries and commands to make sure that data is consistent.



CQRSified

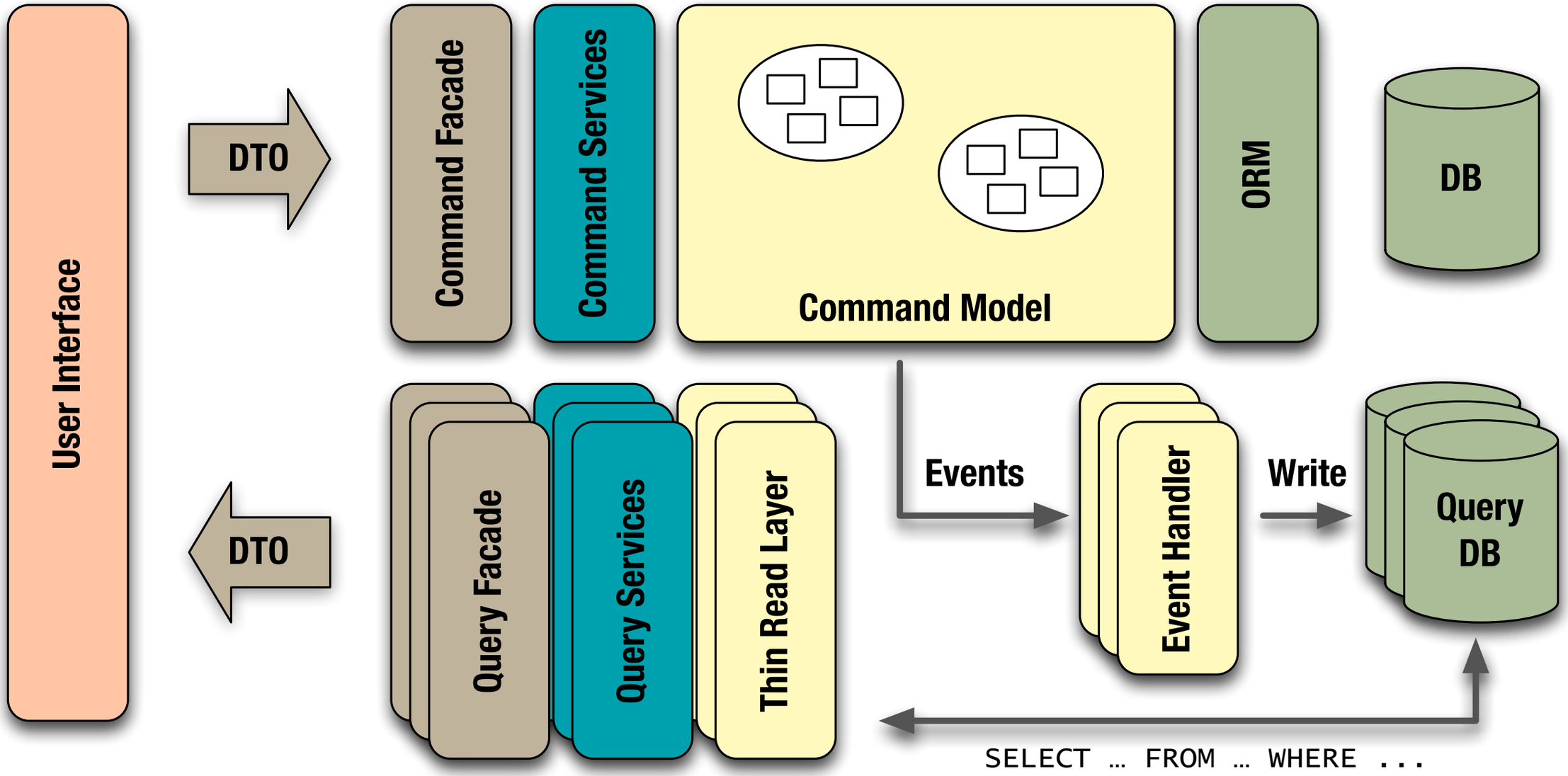
The ~~default~~ architecture for distributed business apps



CQRSified

The ~~default~~ architecture for distributed business apps

In many cases, eventual consistency is sufficient.
The data users are looking at in the UI is always stale to some extent.

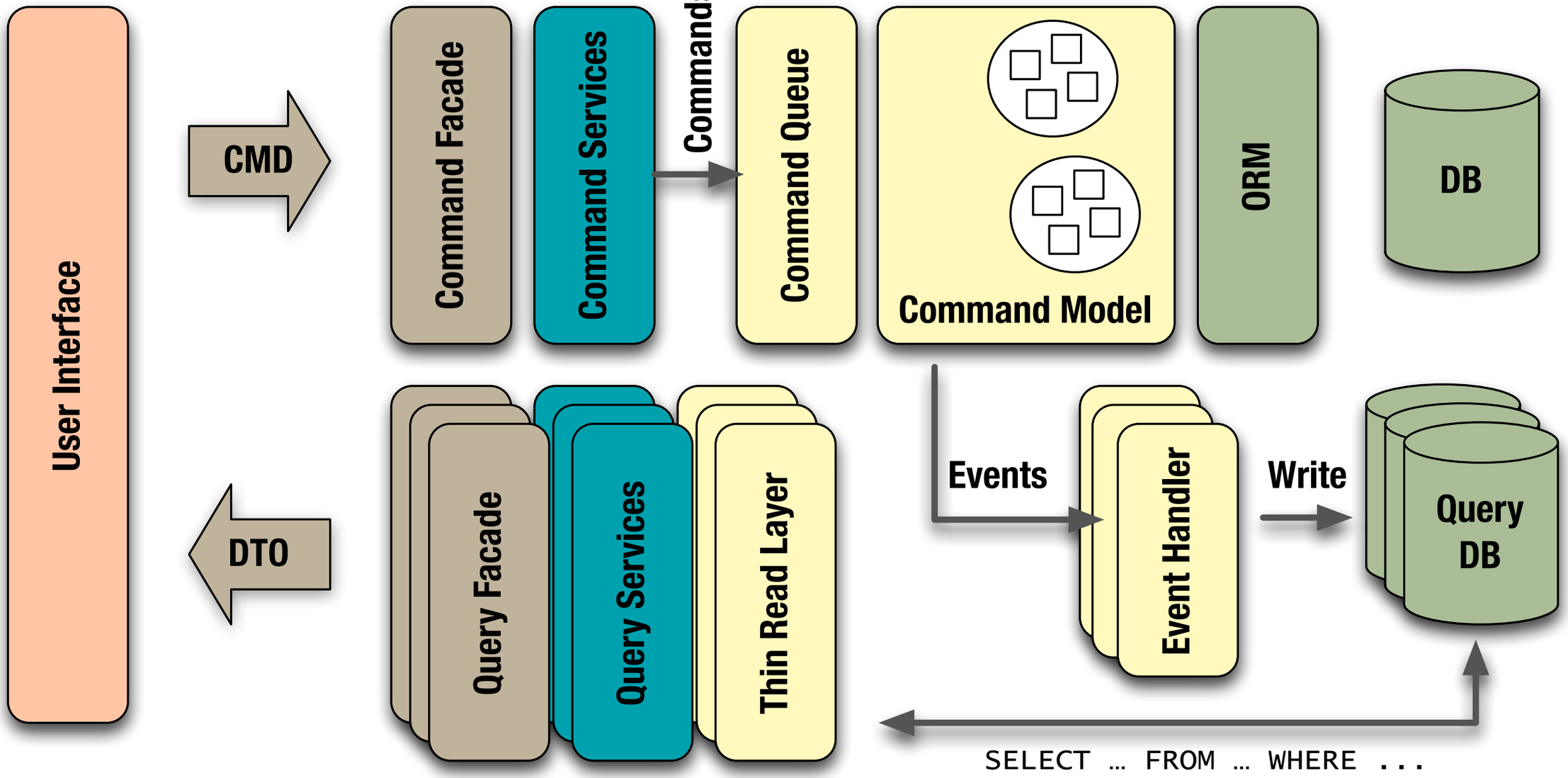


Assumption:
Commands must be processed immediately to ensure data consistency.

Assumption:
Commands must be processed immediately to ensure data consistency.

CQRSified

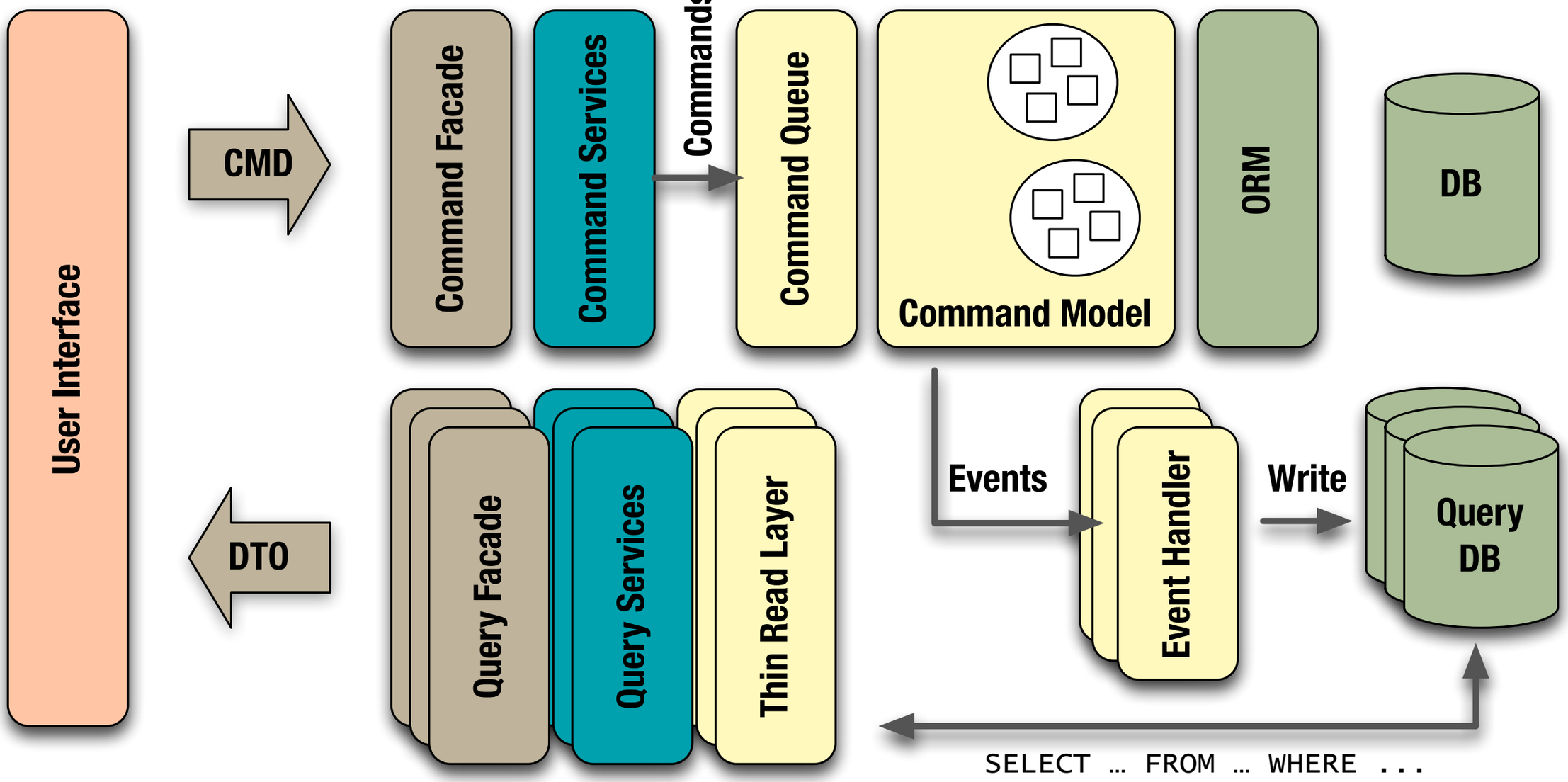
The ~~default~~ architecture for distributed business apps



CQRSified

The ~~default~~ architecture for distributed business apps

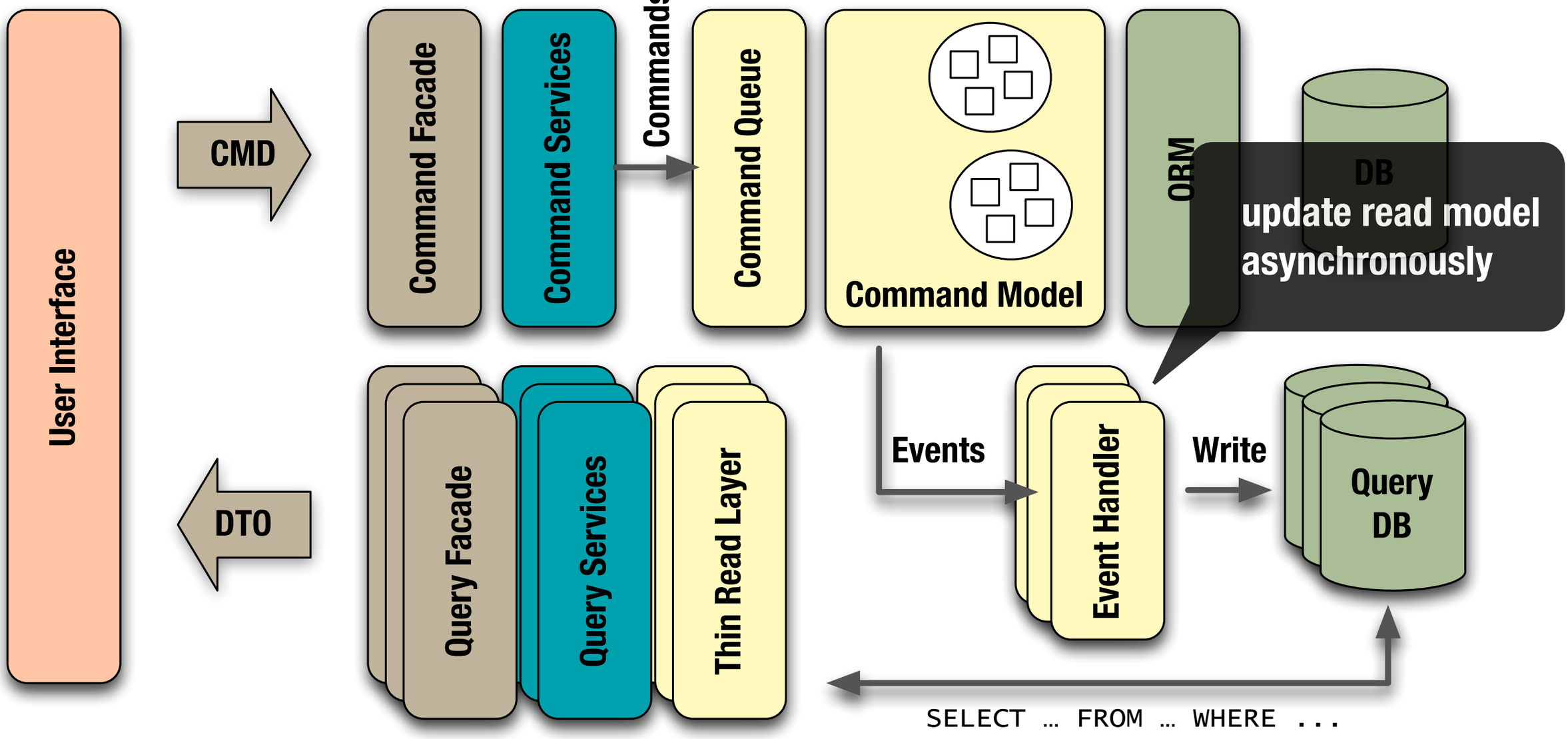
In many cases, users don't care whether their actions have immediate effect – as long as they eventually get feedback.



CQRSified

The ~~default~~ architecture for distributed business apps

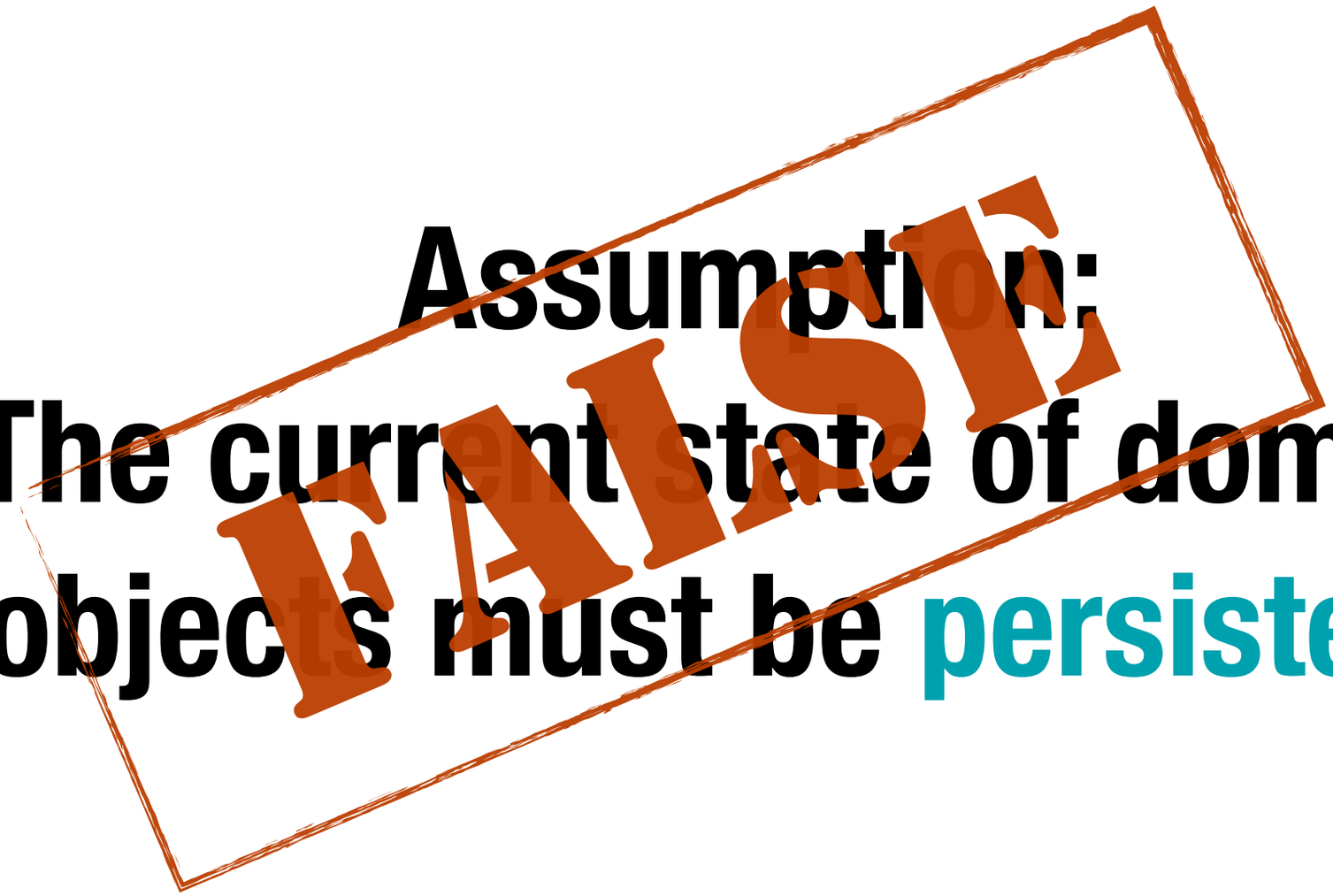
In many cases, users don't care whether their actions have immediate effect – as long as they eventually get feedback.



Assumption:
The current state of domain objects must be persistent.

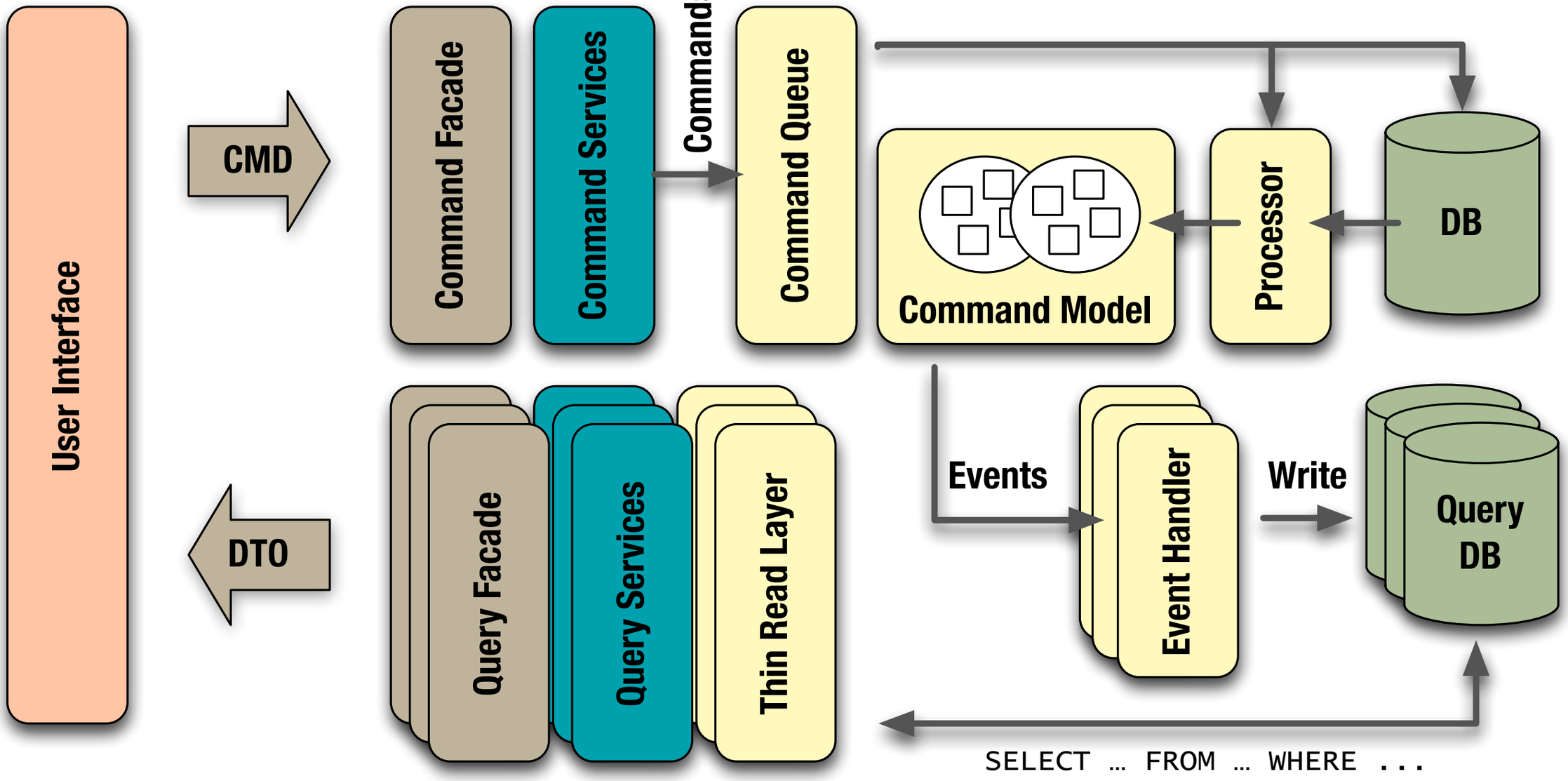
Assumption:

The current state of domain objects must be persistent.



CQRSified

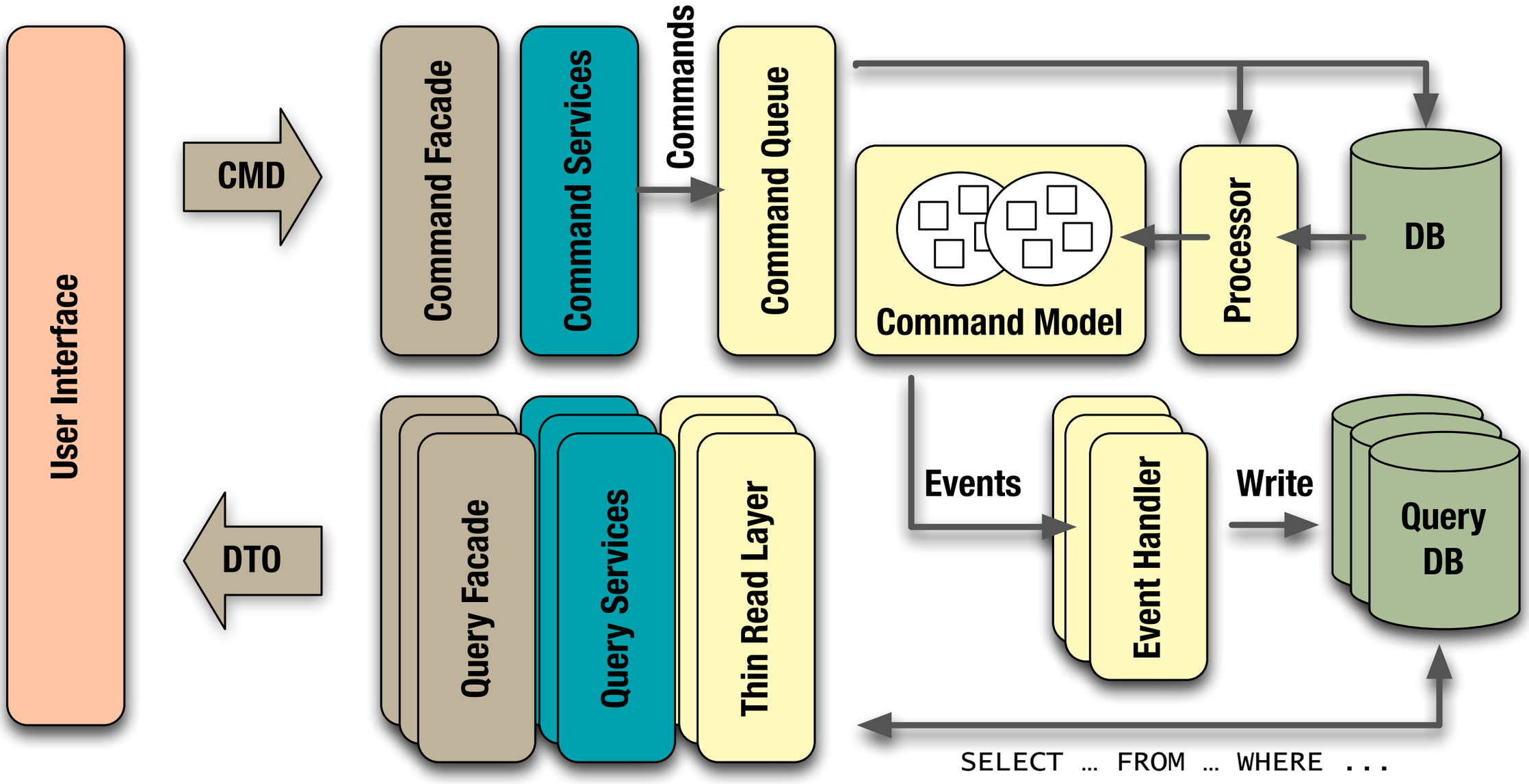
The ~~default~~ architecture for distributed business apps



CQRSified

The ~~default~~ architecture for distributed business apps

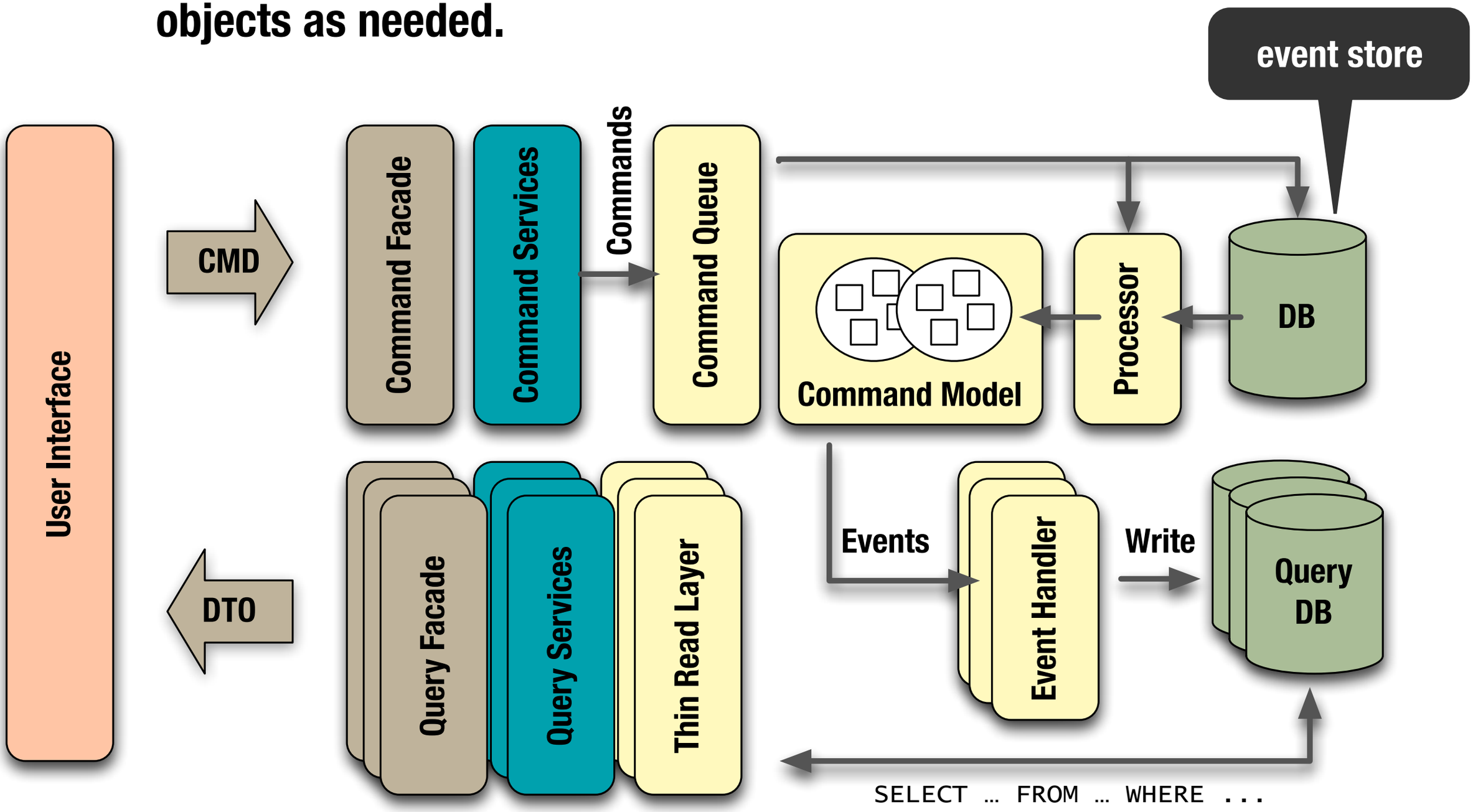
CQRS plays well with an Event Sourcing architecture – you just store events and re-create the state of domain objects as needed.



CQRSified

The ~~default~~ architecture for distributed business apps

CQRS plays well with an Event Sourcing architecture – you just store events and re-create the state of domain objects as needed.



Event Sourcing in a nutshell

Event Sourcing in a nutshell

- capture each update to application state in an event

Event Sourcing in a nutshell

- capture each update to application state in an event
- changes to domain objects are the result of applying events

Event Sourcing in a nutshell

- capture each update to application state in an event
- changes to domain objects are the result of applying events
- events are immutable

Event Sourcing in a nutshell

- capture each update to application state in an event
- changes to domain objects are the result of applying events
- events are immutable
- events are a representation of what has happened at a specific time

What’s so great about Event Sourcing?

What's so great about Event Sourcing?

- allows you to rebuild application state at any point in time just by replaying events up to that point in time

What's so great about Event Sourcing?

- allows you to rebuild application state at any point in time just by replaying events up to that point in time

- allows you to analyse historic data based on detailed events that would otherwise have been lost

What's so great about Event Sourcing?

- allows you to rebuild application state at any point in time just by replaying events up to that point in time
- allows you to analyse historic data based on detailed events that would otherwise have been lost
- gives you an audit log "for free"

What's so great about Event Sourcing?

- allows you to rebuild application state at any point in time just by replaying events up to that point in time
- allows you to analyse historic data based on detailed events that would otherwise have been lost
- gives you an audit log "for free"
- you can add new, optimized read models (potentially in-memory) later without migration hassle and such

http://www.flickr.com/photos/kaptainkobold/5170454747/

You probably don’t.

CQRS is not a silver bullet and doesn't apply everywhere.

Beware of the added complexity!

Don’t do CQRS…

Don’t do CQRS…

…if your application is just a simple CRUD-style app.

Don’t do CQRS…

…if your application is just a simple CRUD-style app.

…if you don’t have scaling issues.

Don’t do CQRS…

…if your application is just a simple CRUD-style app.

…if you don’t have scaling issues.

…if it doesn’t improve your domain models.

Consider doing CQRS…

…if your write/read load ratio is highly asymmetrical.

…if scaling your application is difficult.

…if your domain model is bloated by complex domain logic, making queries inefficient.

…if you can benefit from event sourcing.

Frameworks, any?

Axon Framework (Java)

www.axonframework.org

Axon Framework (Java)

www.axonframework.org

Lokad (.NET)

http://lokad.github.com/lokad-cqrs/

Axon Framework (Java)

www.axonframework.org

Lokad (.NET)

http://lokad.github.com/lokad-cqrs/

…and some more

Piotr Wyczesany

DDD/CQRS/ES in practice – Axon to the rescue

Today, 11:40am
Room 1

That’s it. Really.

Feel free to ask me anything!

@owolf