

CQRS for Great Good **(Neue Wege mit CQRS)**

Oliver Wolf



Oliver Wolf
@owolf



www.innoQ.com
@innoQ

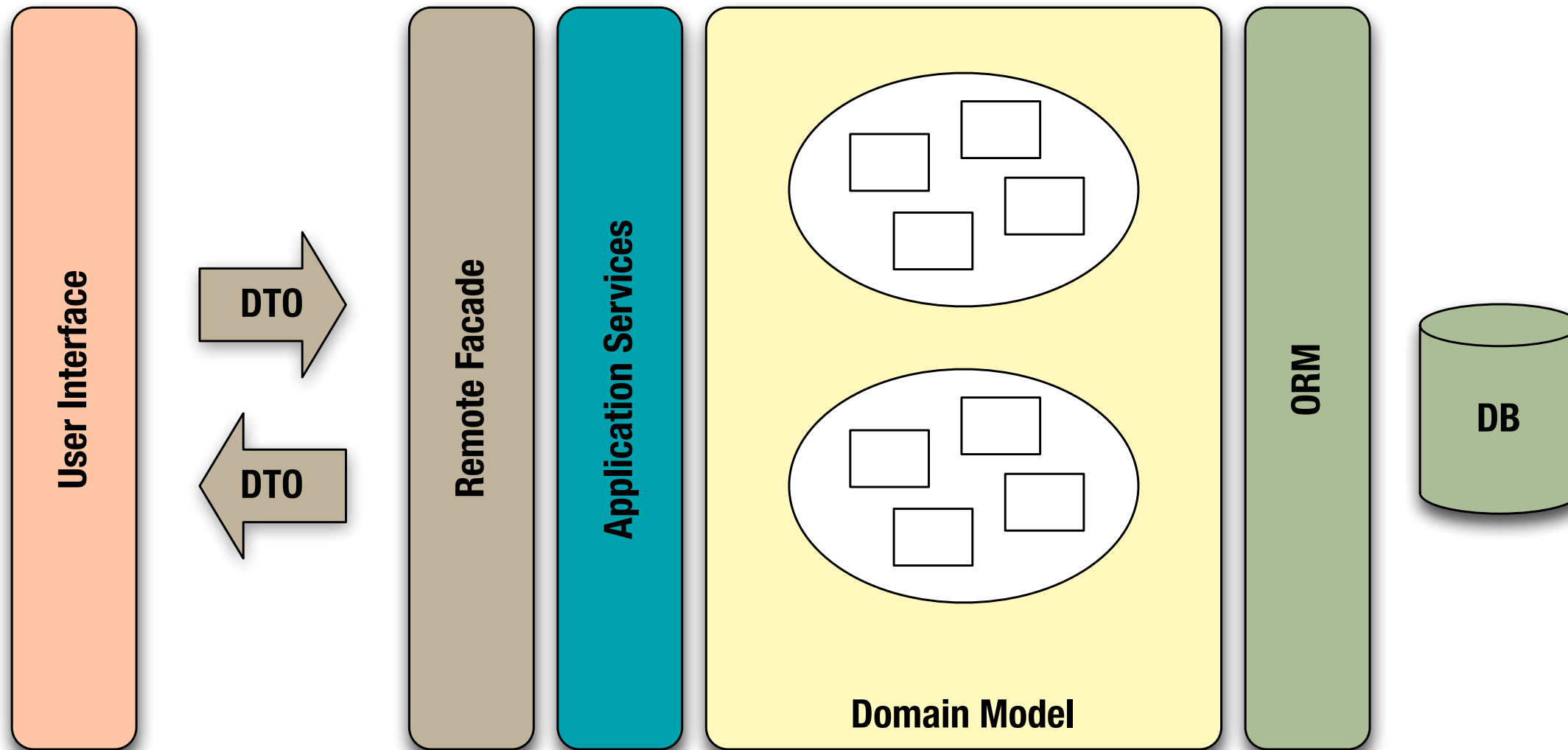
CQRS

Command Query Responsibility Segregation

The default architecture for distributed business apps

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

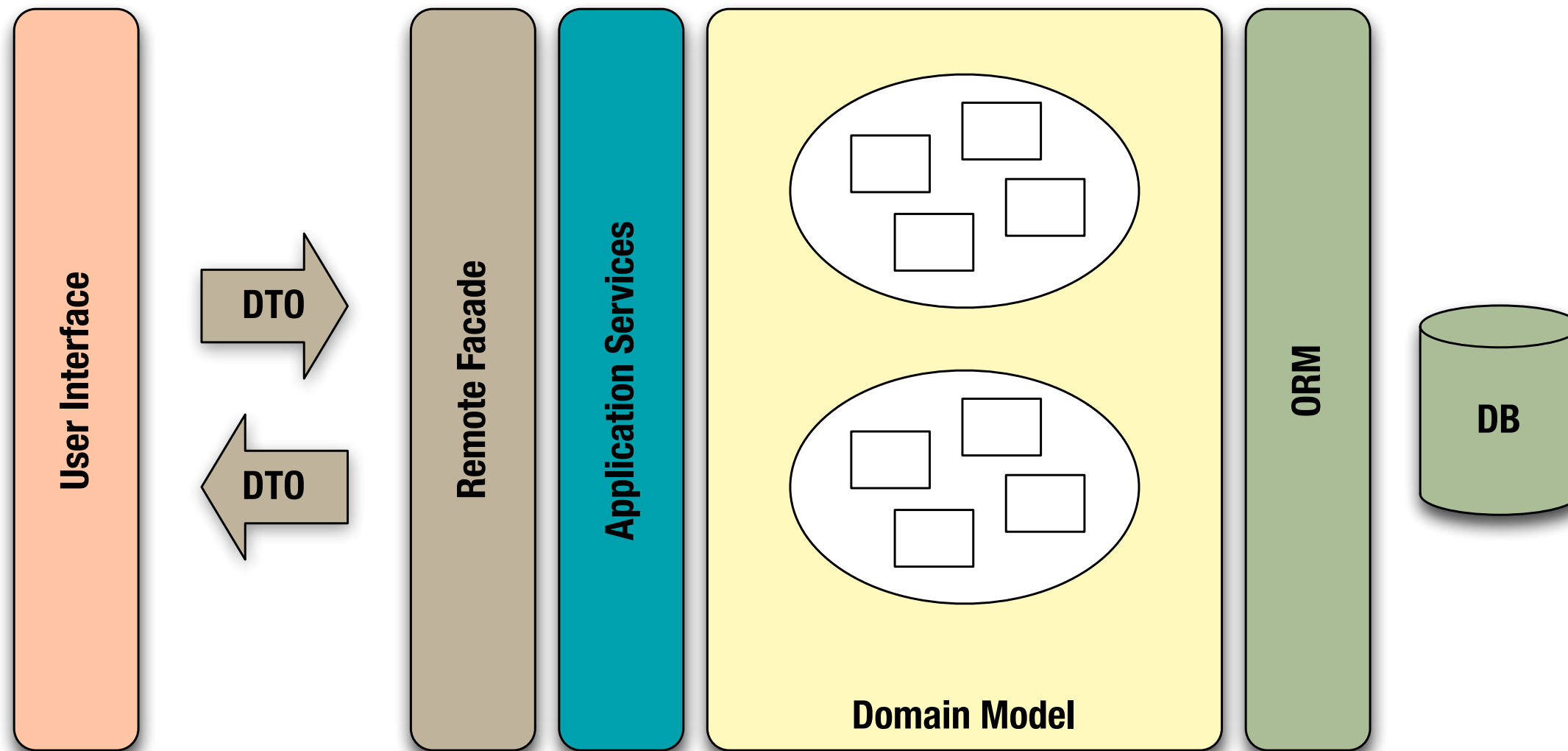
```
findCustomers()  
getCustomer()  
updateCustomer()
```



The default architecture for distributed business apps

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

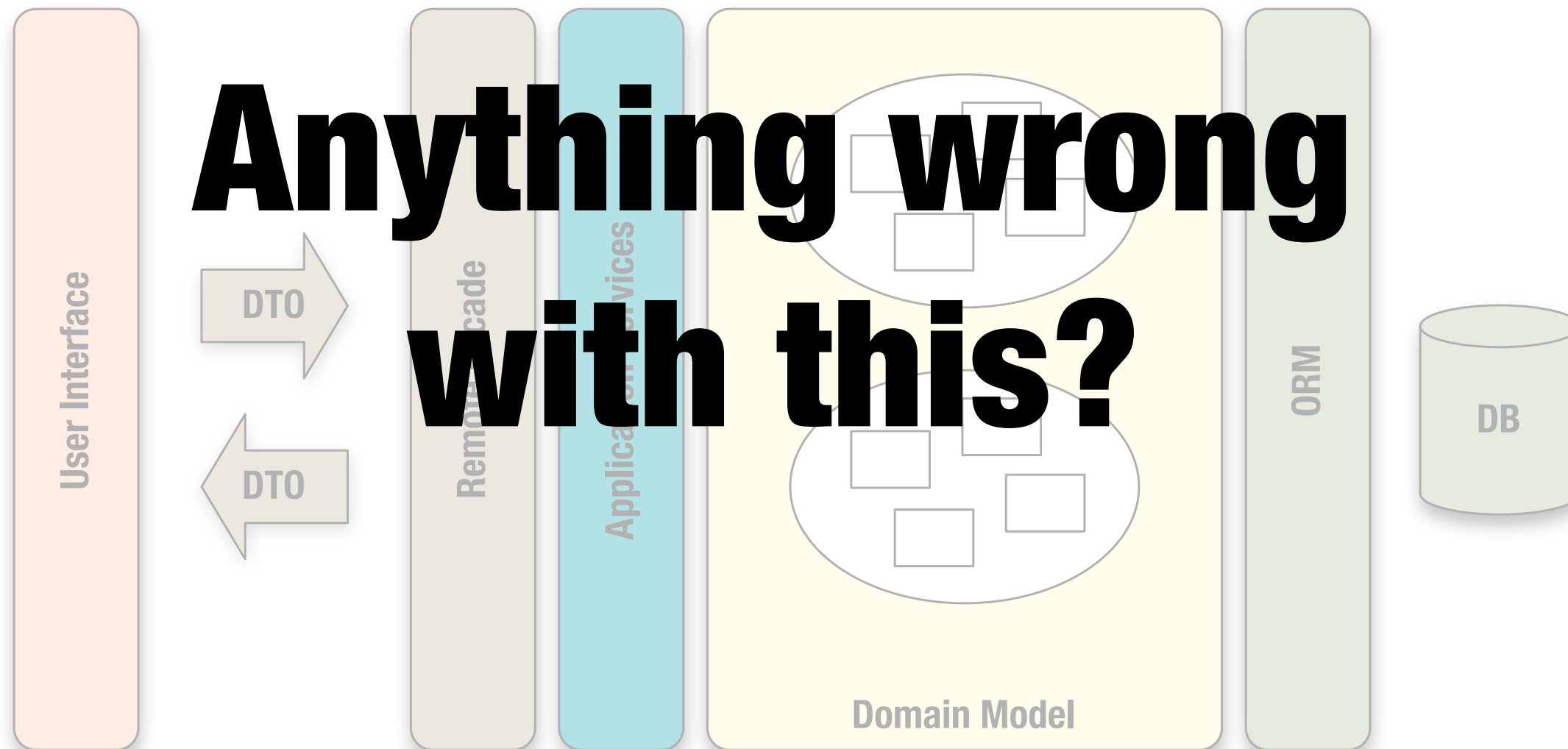
```
GET /customers?filter=...  
GET /customer/{id}  
PUT /customer/{id}
```



The default architecture for distributed business apps

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

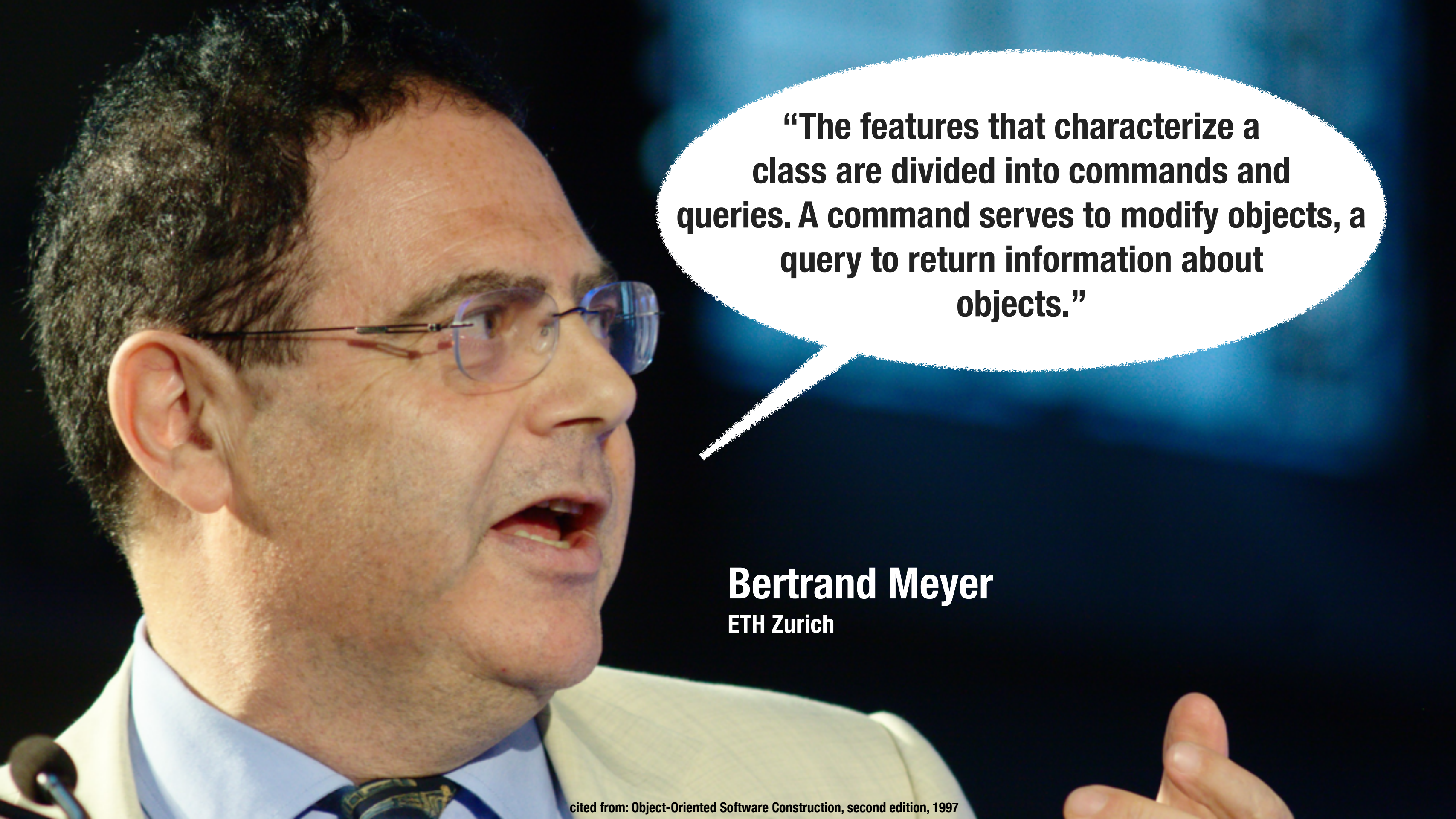
```
GET /customers?filter=...  
GET /customer/{id}  
PUT /customer/{id}
```



Maybe not.

Scalability?

Domain Model?

A close-up, profile view of Bertrand Meyer, a man with dark hair and glasses, wearing a light blue shirt and a patterned tie. He is speaking, with his mouth open. The background is dark and out of focus.

“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.”

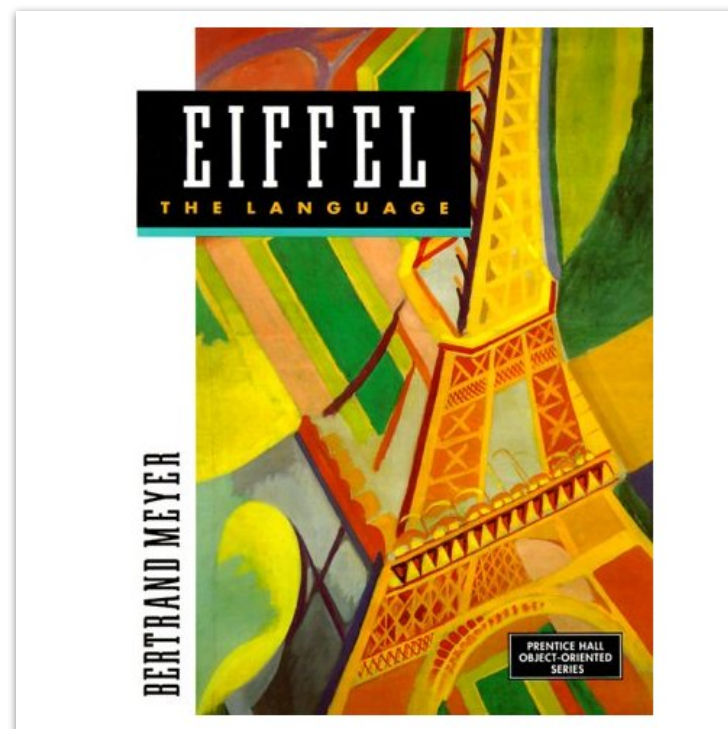
Bertrand Meyer
ETH Zurich

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();  
}
```

Mutates state

**Returns a value without
causing side effects**

**Scope is a Bounded
Context**

CQRS

=

CQS in the large

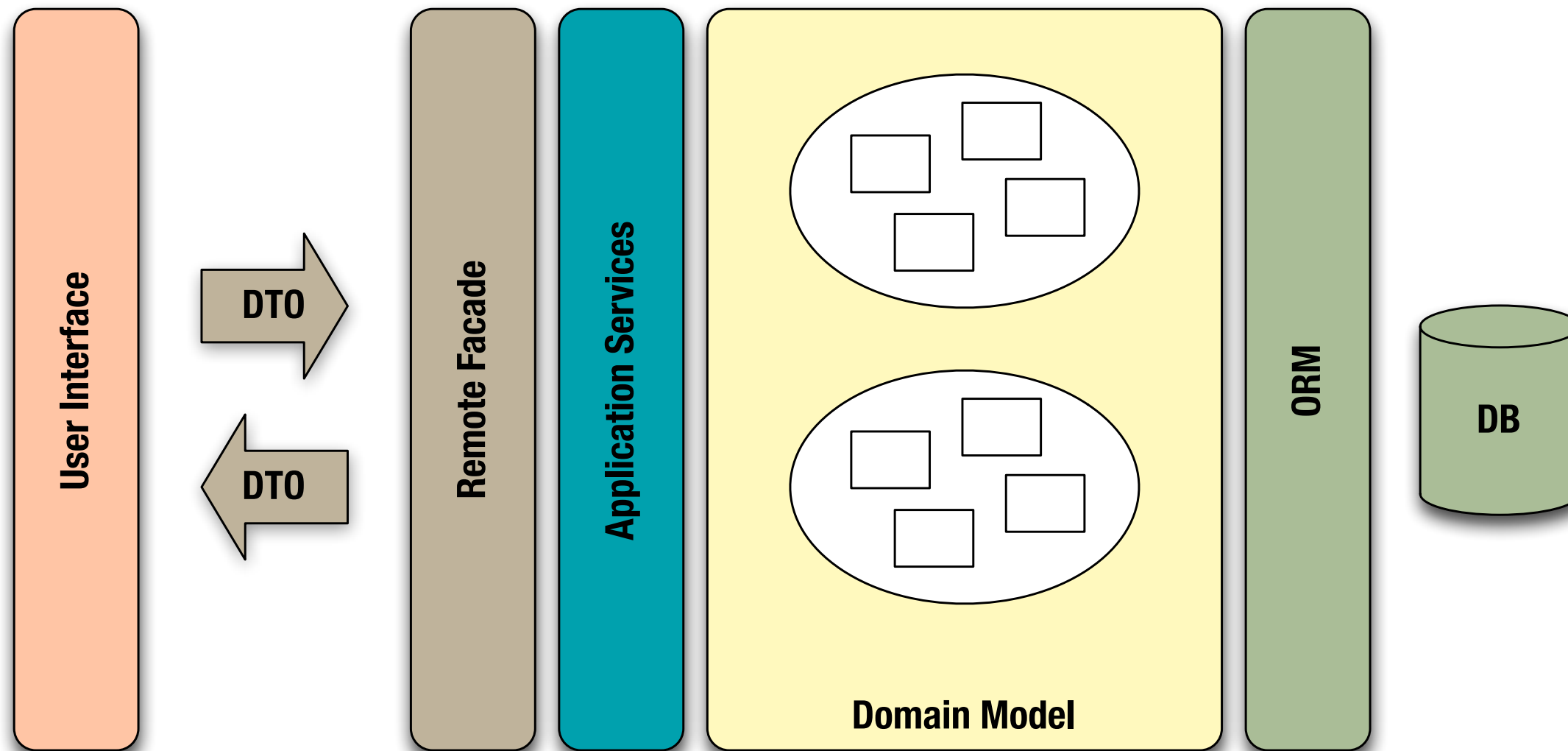
**Scope is a single
class**

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

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

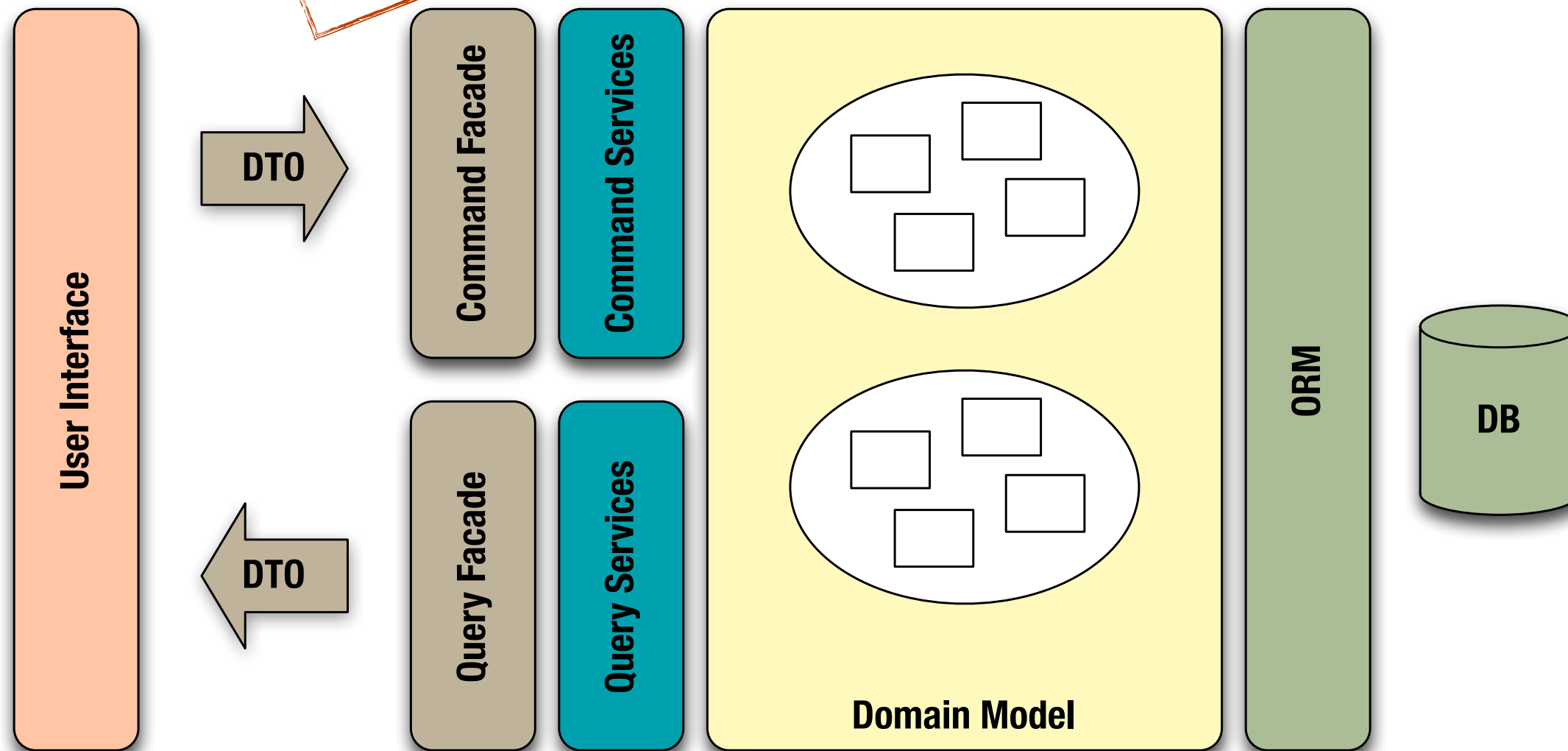
```
interface CustomerCommandService {  
    void updateCustomer(Customer);  
    void deleteCustomer(ID);  
}
```

The default architecture for distributed business apps



The default architecture for distributed business apps

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 encourages you to
challenge established
assumptions and opens up new
architectural options!**

**Unlearn what
you have
learned you
must.**



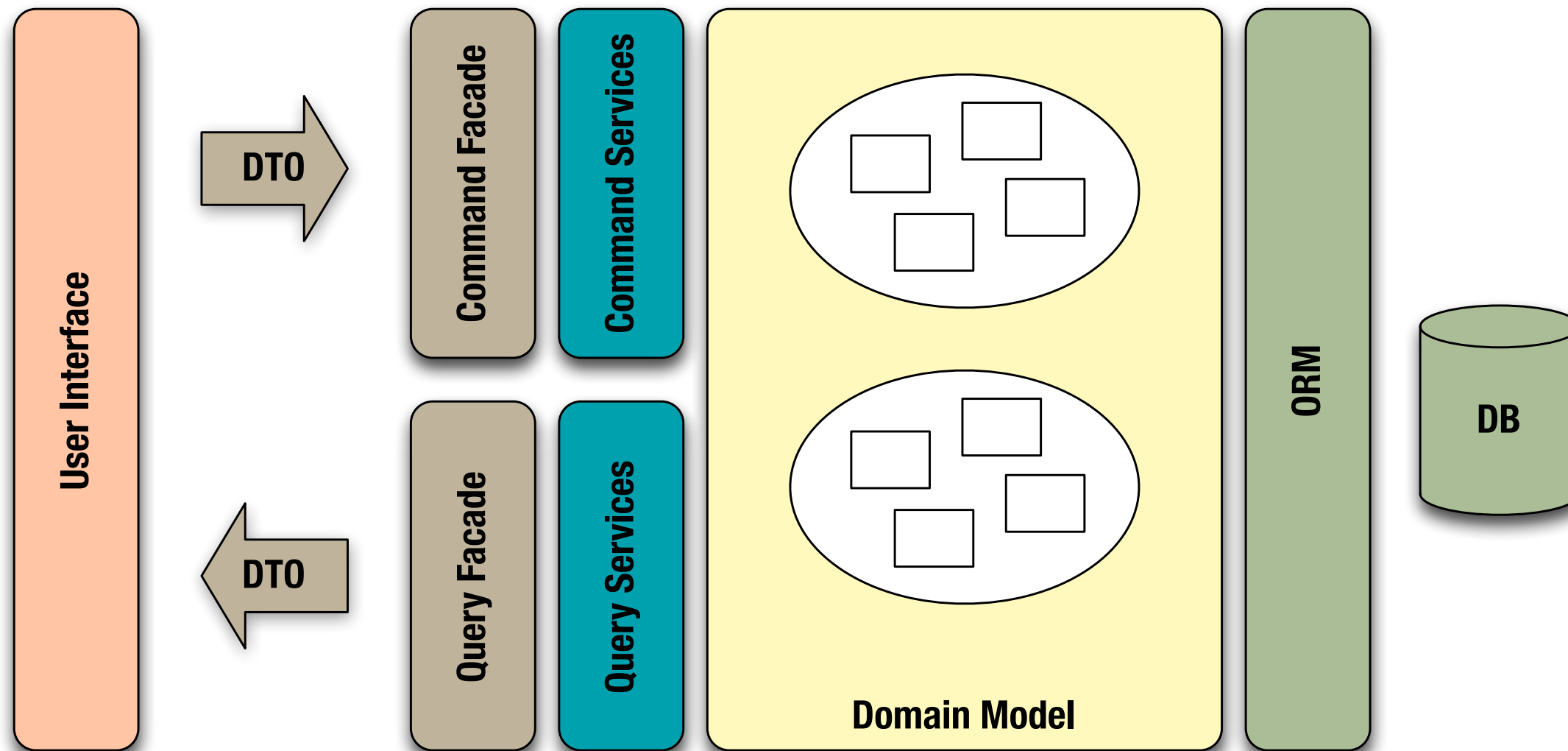


assumptions
we often take for granted

Assumption 1:
Reads and writes are strongly
cohesive, so they must be part of the
same Bounded Context.

Assumption 1:
Reads and writes are strongly
cohesive, so they must be part of the
same Bounded Context.

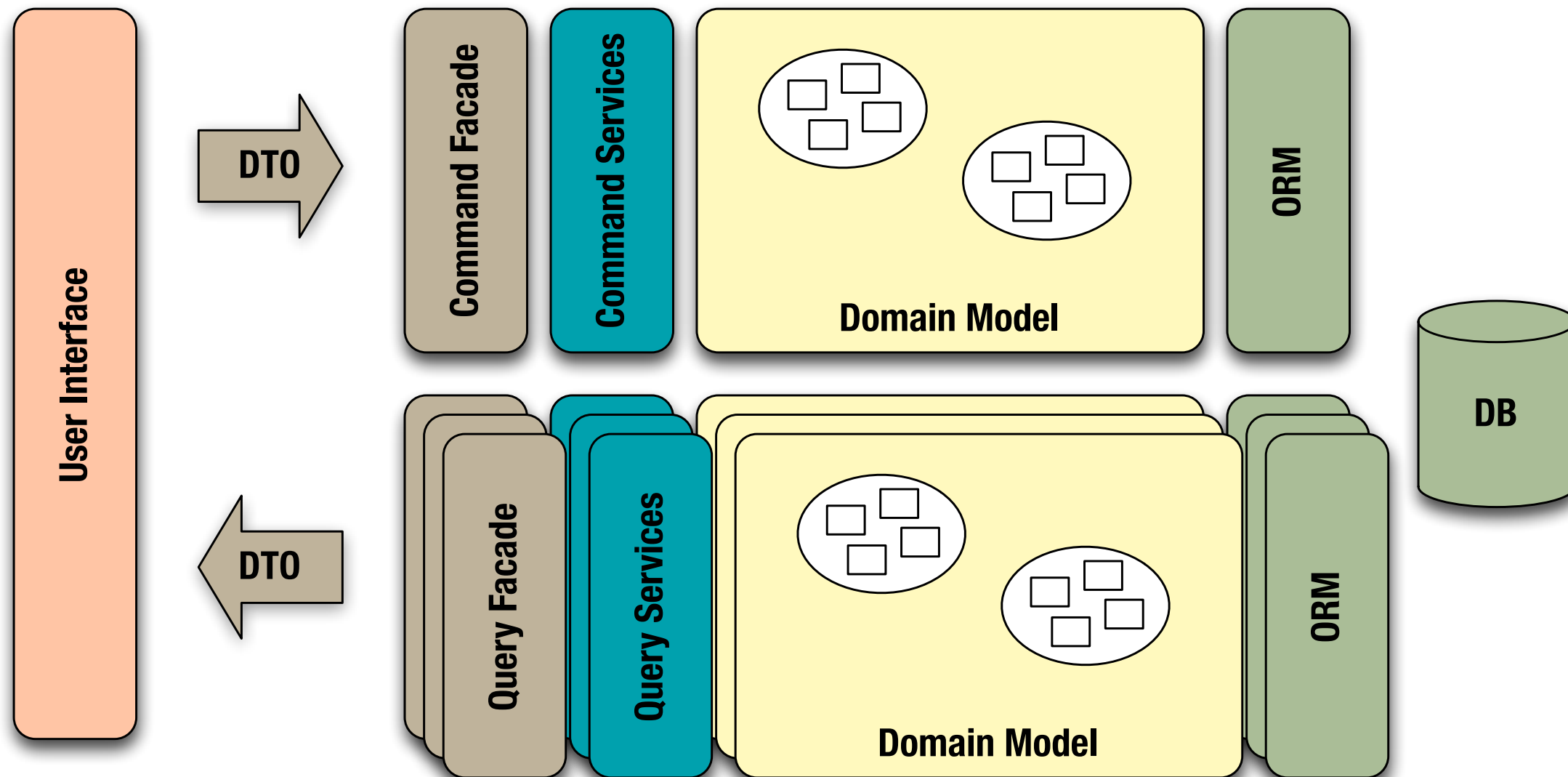
~~CQRSified~~ The default architecture for distributed business apps



CQRSified

The ~~default~~ architecture for distributed business apps

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



Assumption 2:

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

Assumption 2:

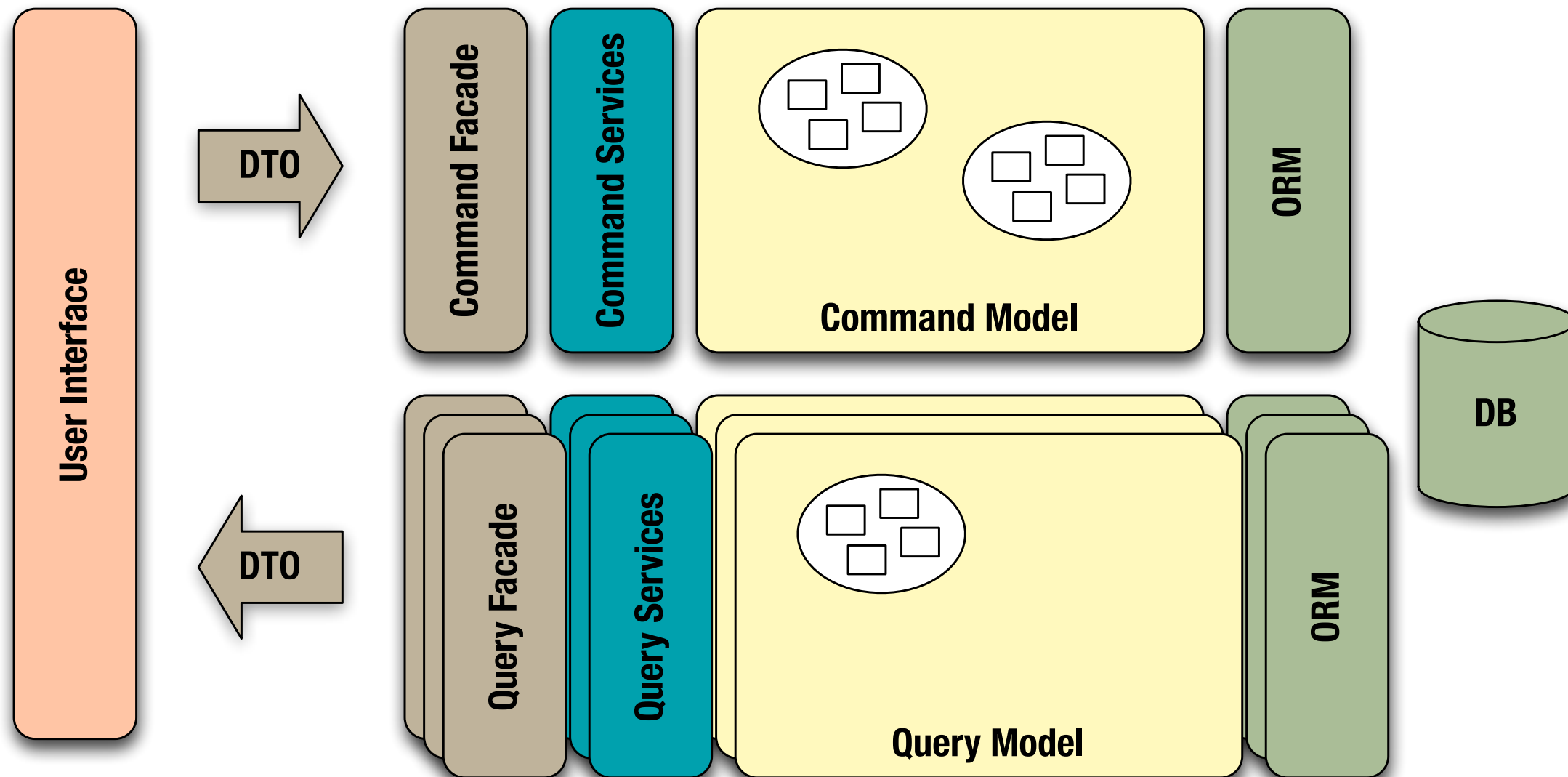
Reads and writes use the same data, so they must be served from and applied to the same domain model.

FALSE!

CQRSified

The ~~default~~ architecture for distributed business apps

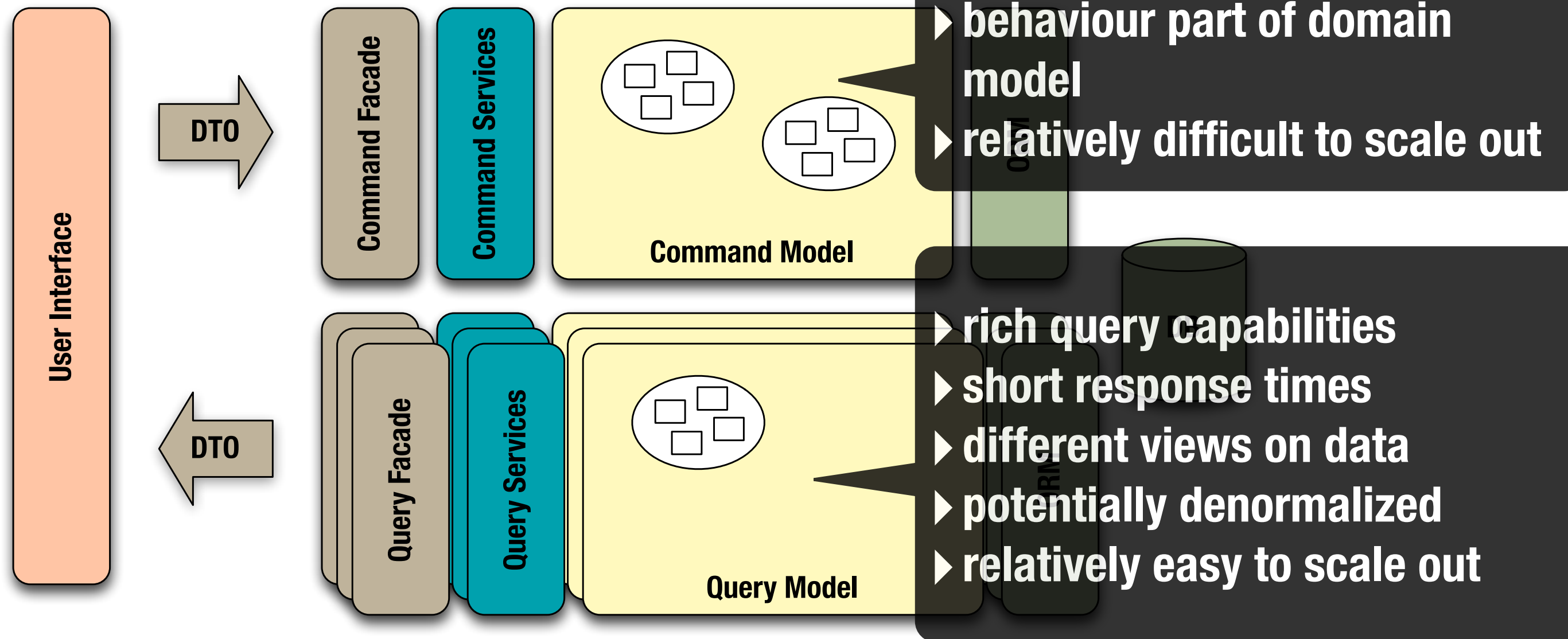
Queries can benefit from a specialized query model, optimized for quick data retrieval (de-normalized, pre-aggregated,...)



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 3:

Even for queries, we have to go through the domain model to abstract from the underlying database model.

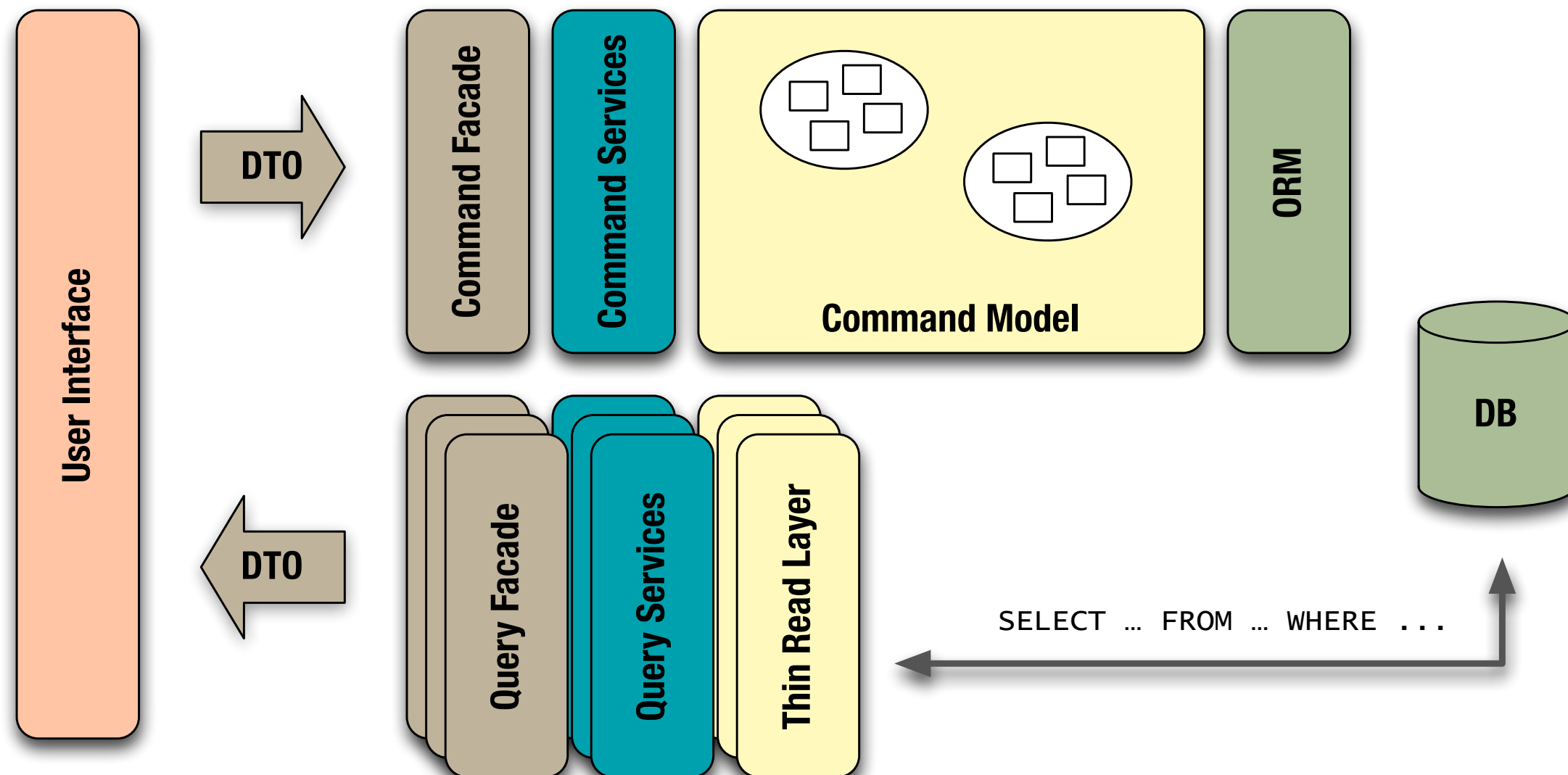
Assumption 3:

Even for queries, we have to go through the domain model to abstract from the underlying database model.

CQRSified

The ~~default~~ architecture for distributed business apps

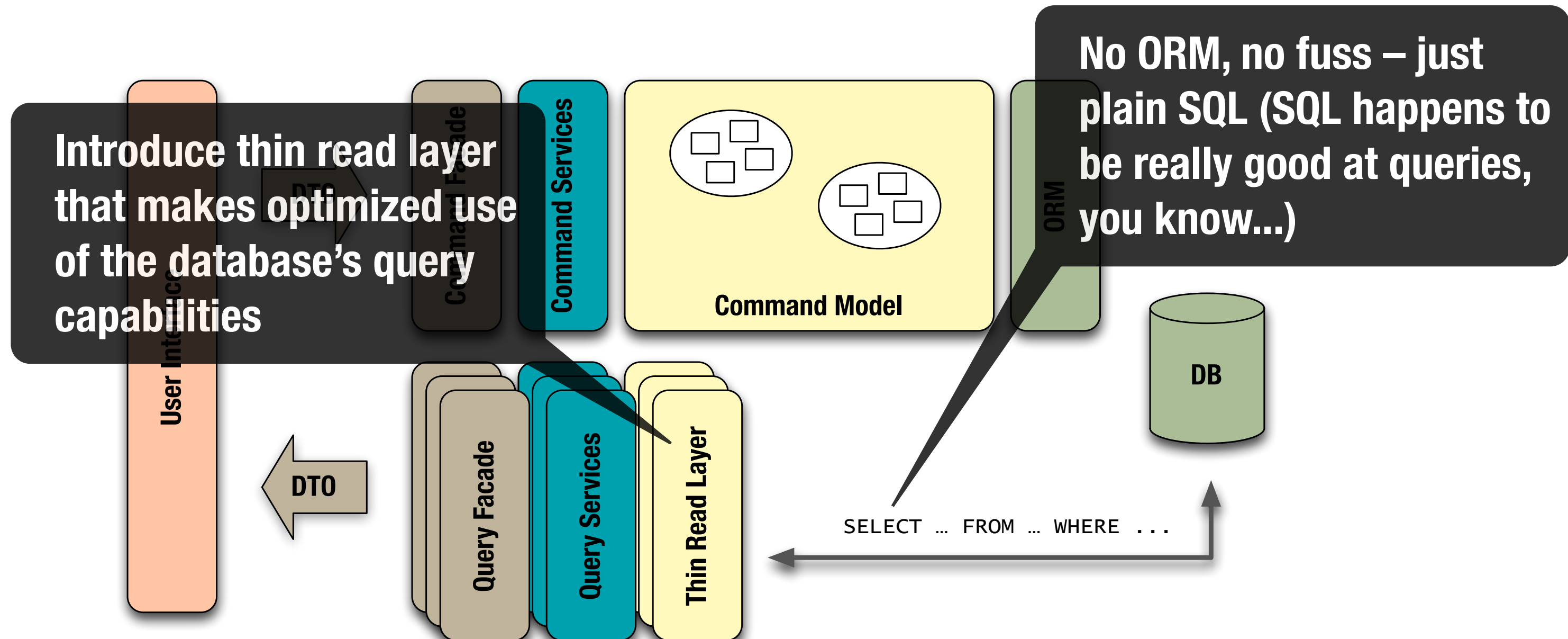
Queries are just dealing with data, not with behaviour.
What do we need objects for?



CQRSified

The ~~default~~ architecture for distributed business apps

Queries are just dealing with data, not with behaviour.
What do we need objects for?



Assumption 4:

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

Assumption 4:

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

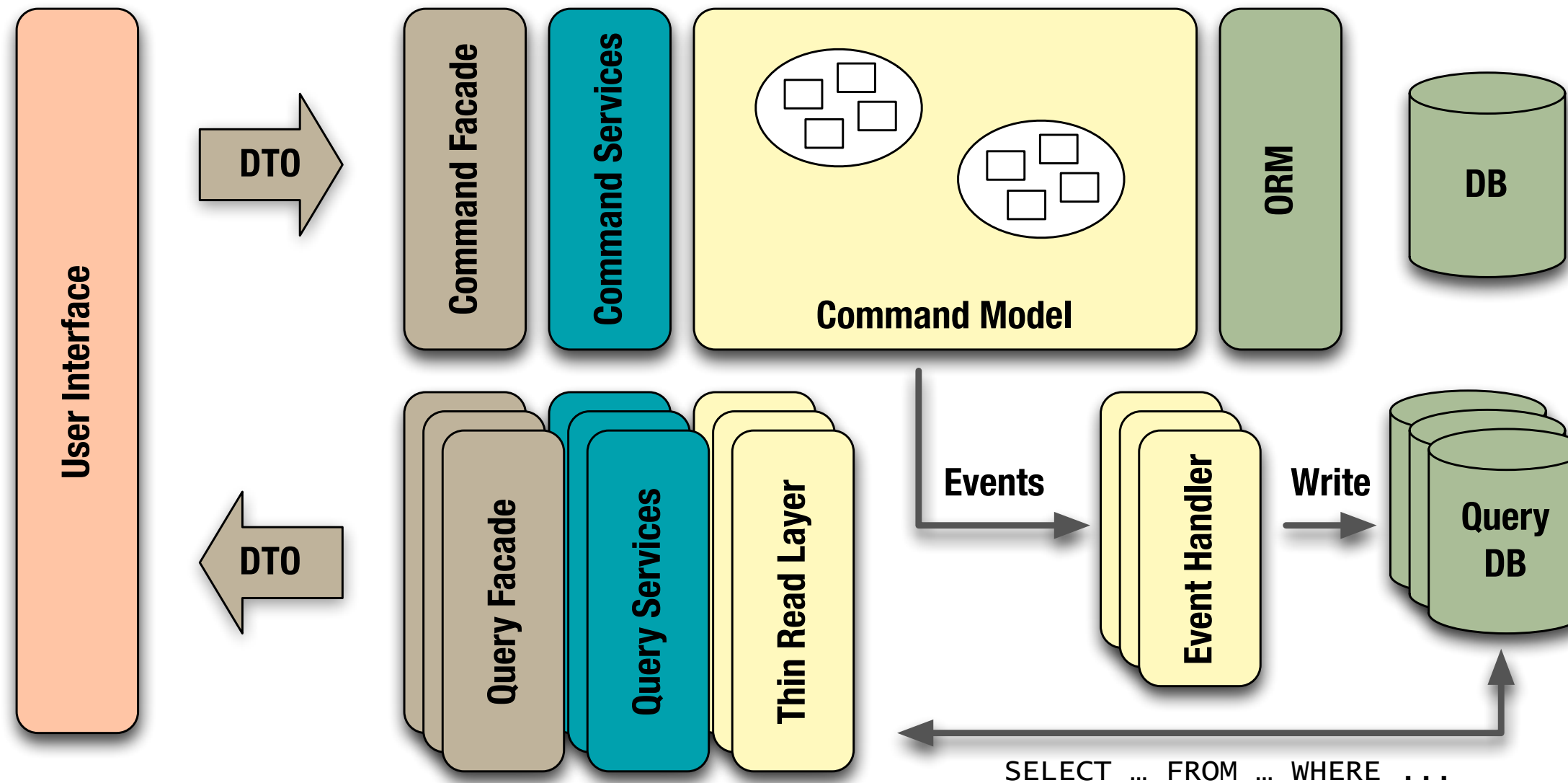
FALSE

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.

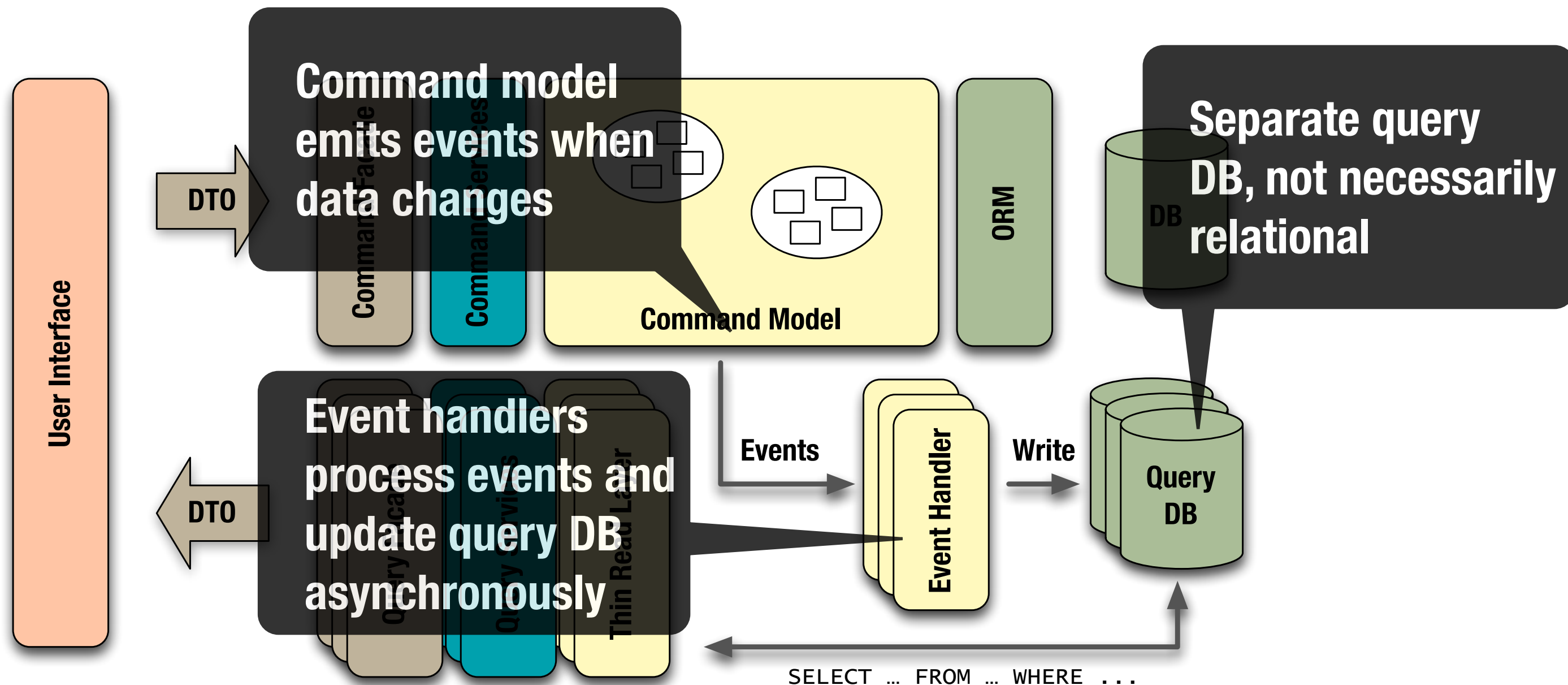


CQRSified

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 5:
Commands must be processed
immediately to ensure data
consistency.

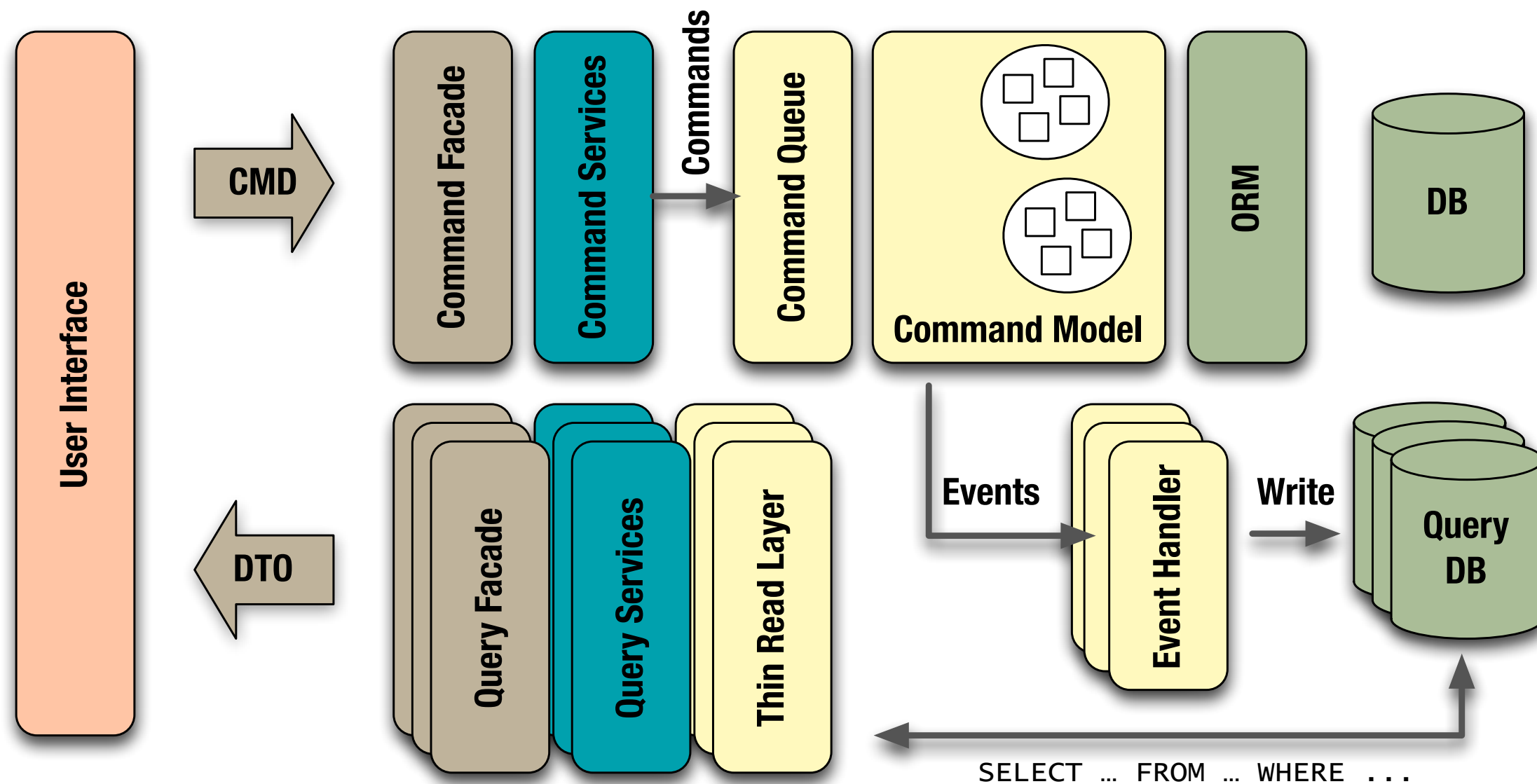
Assumption 5:
Commands must be processed
immediately to ensure data
consistency.

FALSE!

CQRSified

The ~~default~~ architecture for distributed business apps

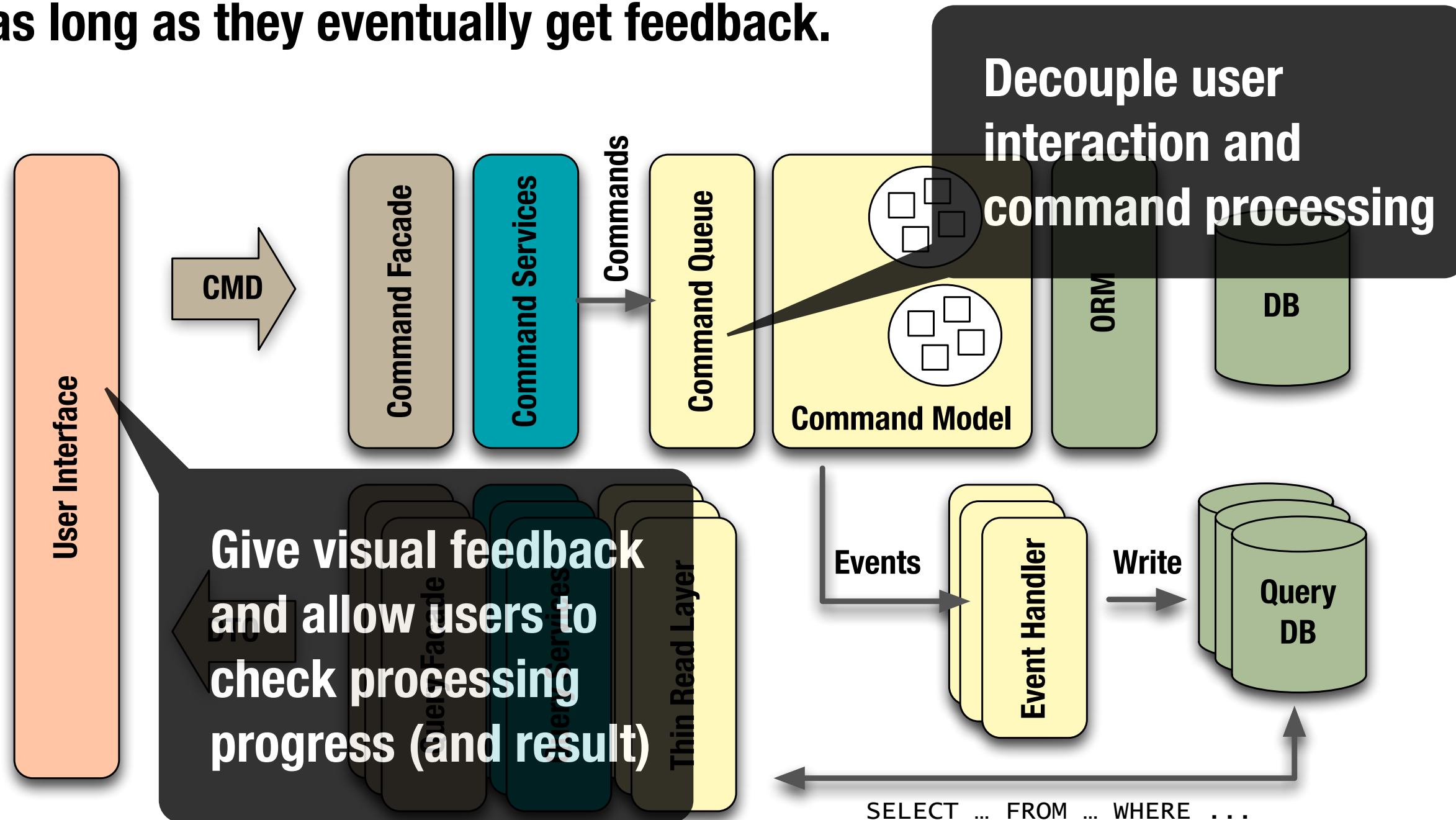
In many cases, users don't care if their actions have immediate effect – as long as they eventually get feedback.



CQRSified

The ~~default~~ architecture for distributed business apps

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



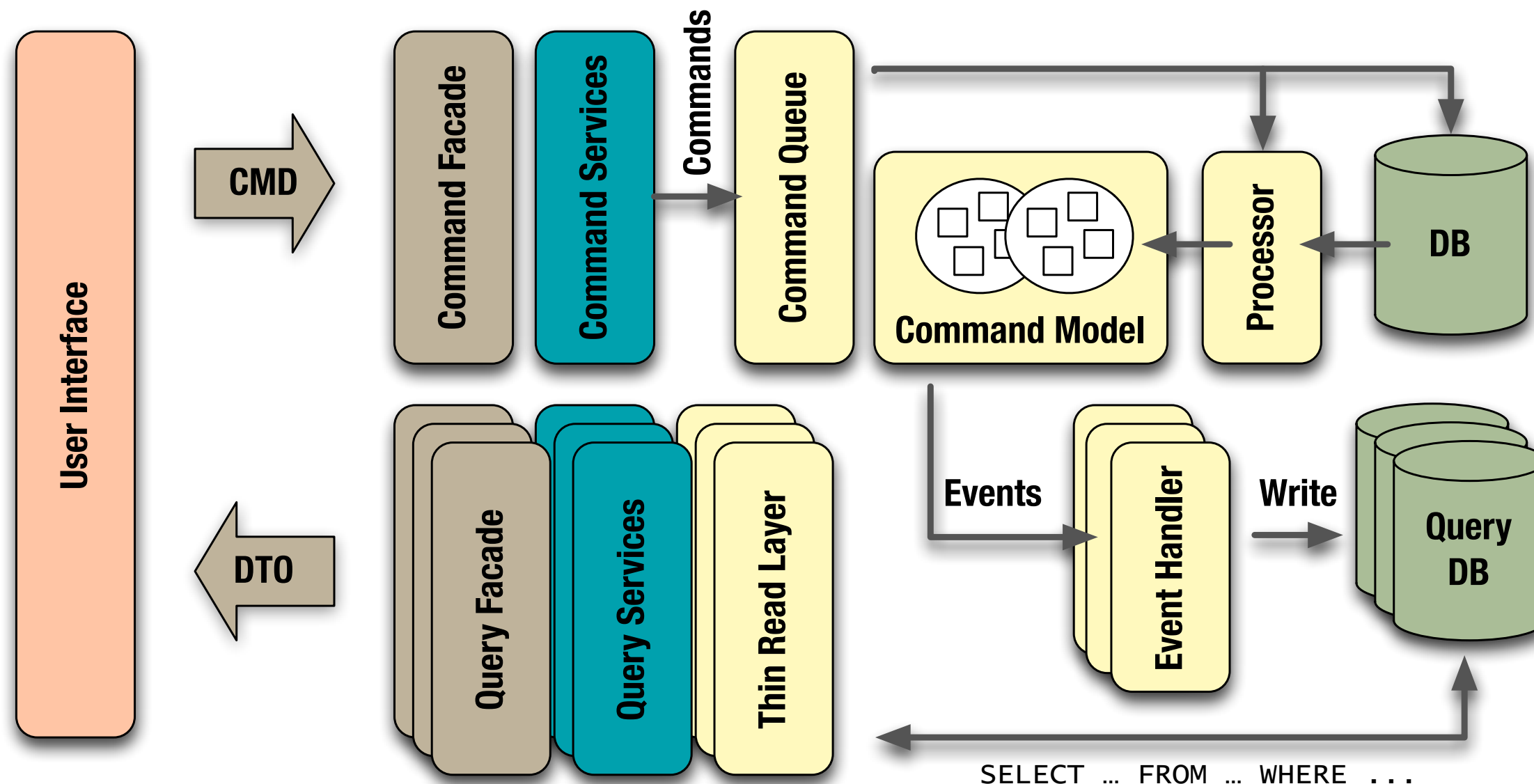
Assumption 6:
The current state of domain objects
must be persistent.

Assumption 6:
The current state of domain objects
must be persistent.

CQRSified

The ~~default~~ architecture for distributed business apps

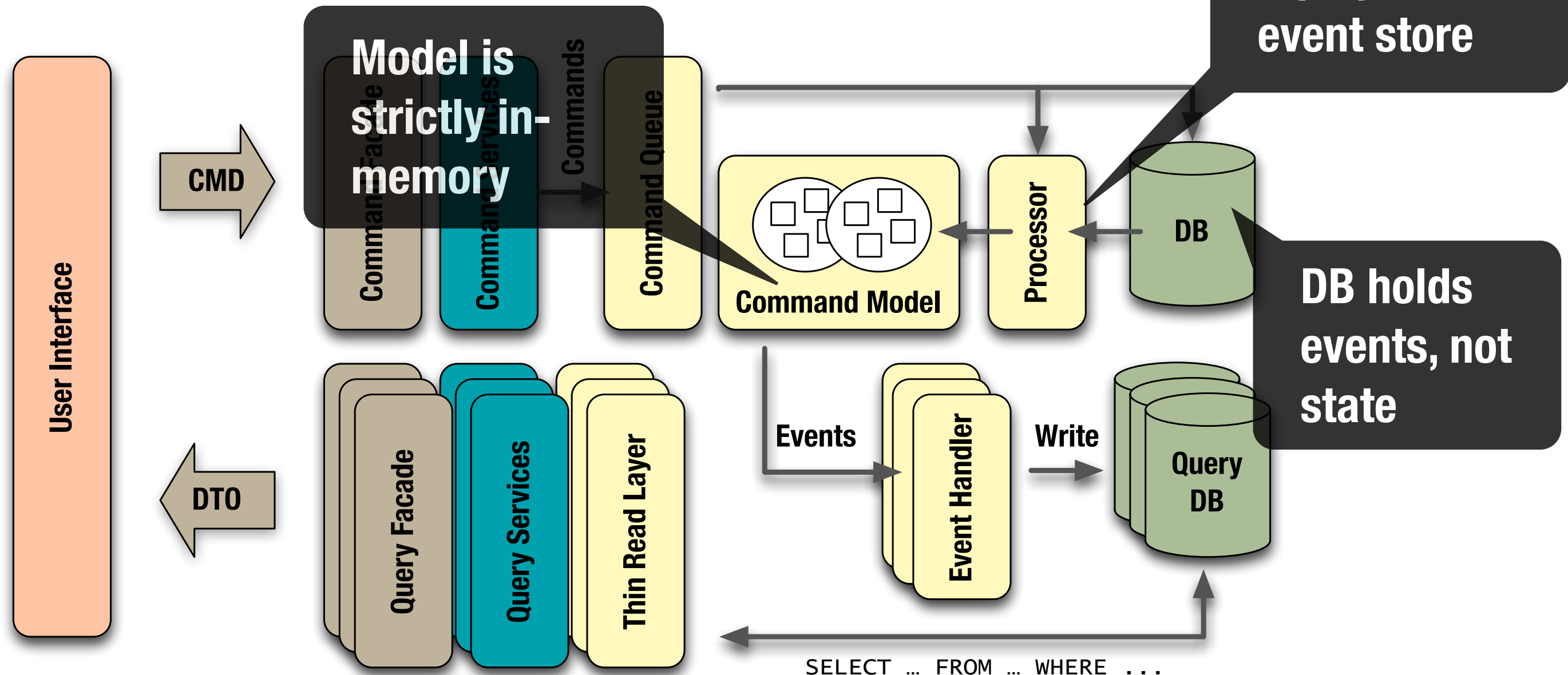
CQRS plays well with an **Event Sourcing** architecture –
you just store events and re-create the state of domain objects as needed.



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

- ▶ **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?

- ▶ allows you to rebuild application state **at any point in time** just by **replaying events** from to 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

**How do I convince my
boss that we'll have to
rewrite our application
with CQRS???**



Chances are you don't.

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

Beware of the added complexity!

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 help 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

Lokad (.NET)

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

...and some more



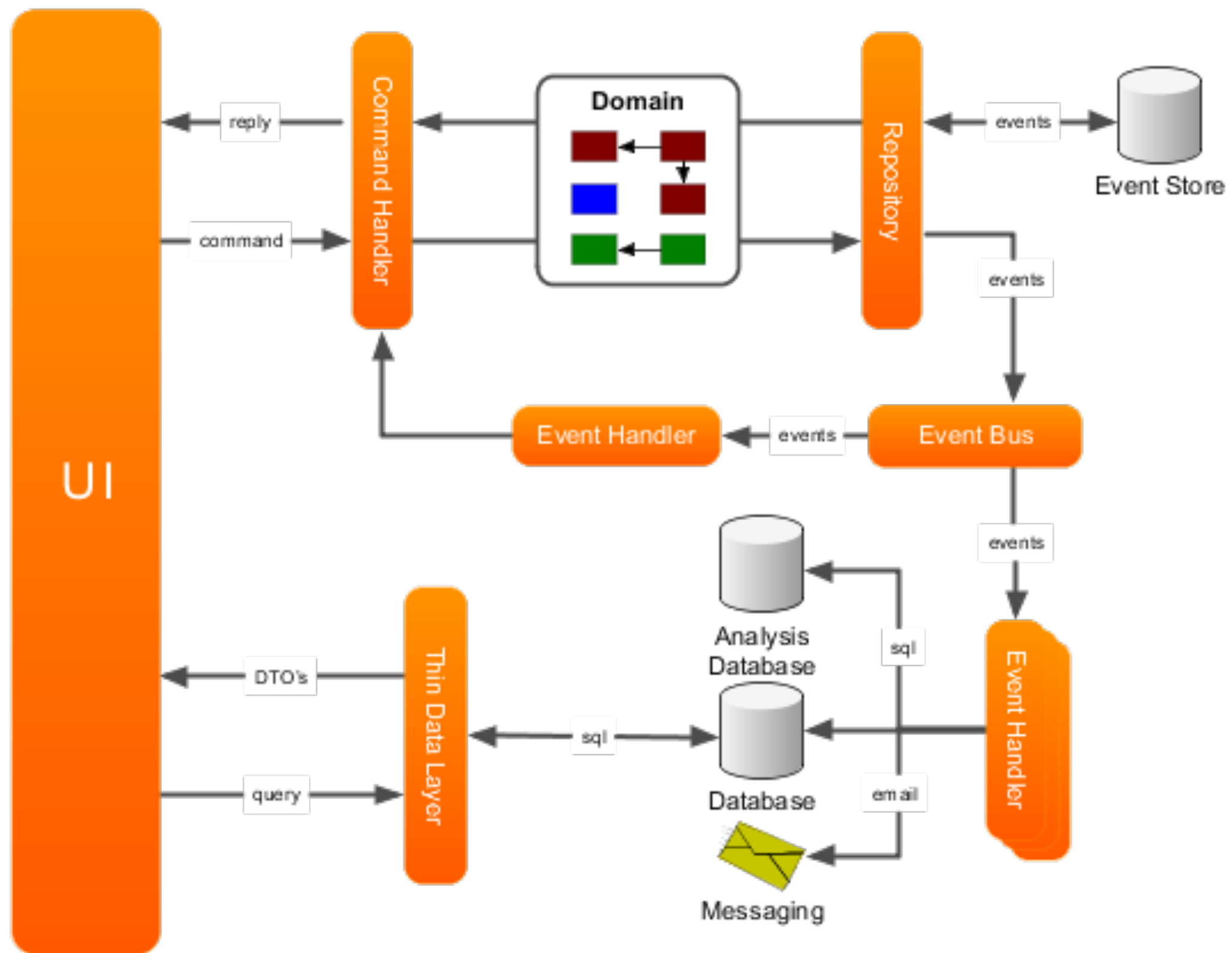
www.axonframework.org

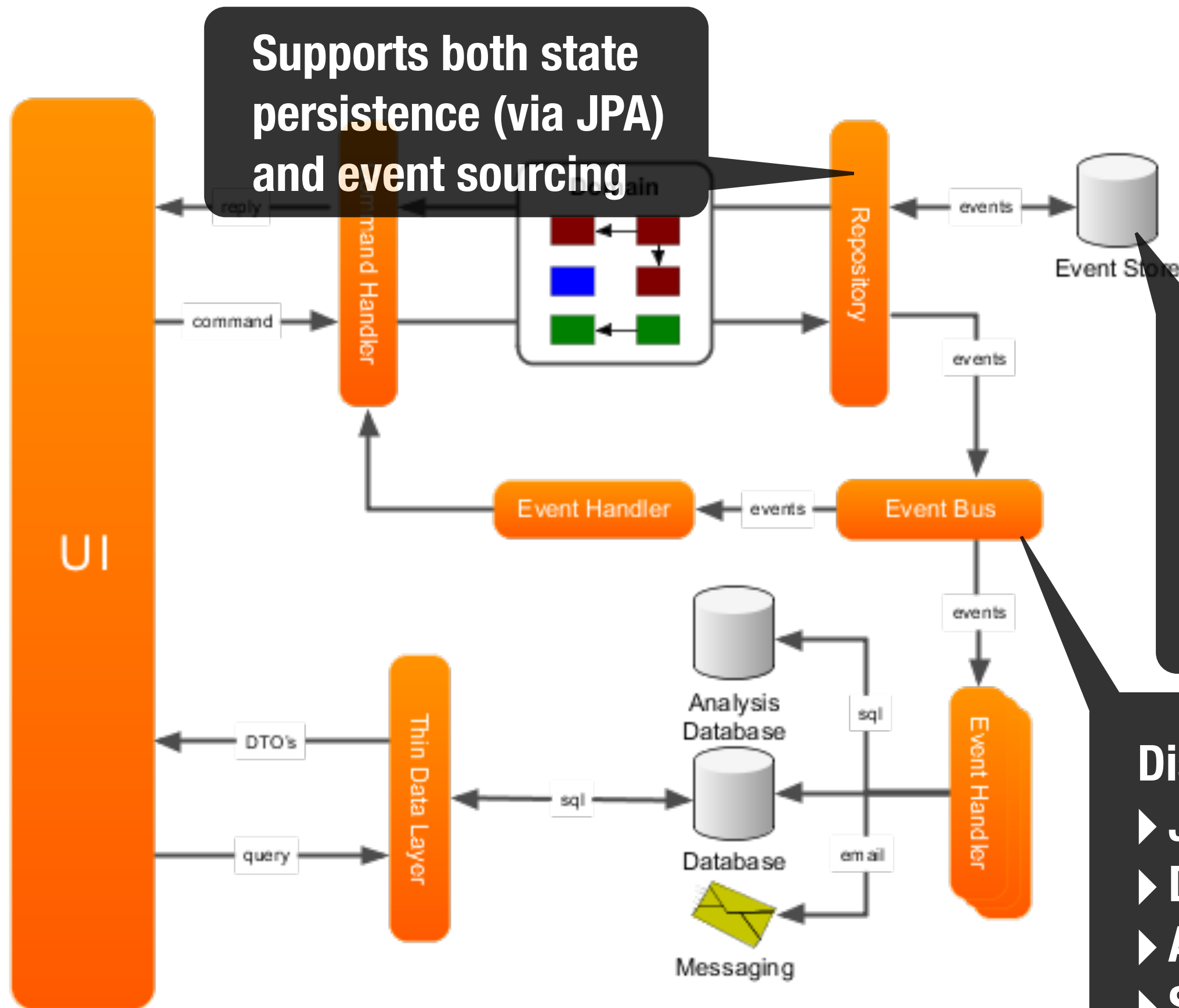
Apache 2 license

Version 2.0 released earlier this year

Maintained by Trifork Netherlands

Commercial support offering available





Supports both state persistence (via JPA) and event sourcing

- File system
- JPA
- MongoDB
- Cassandra
- Redis
- Google App Engine DataStore

- Distributed Event Bus, based on
- JGroup
 - Disruptor
 - AMQP
 - Spring Integration

Things I like about Axon

- ▶ **relatively unobtrusive, commands and events are just Java objects**
- ▶ **Event sourcing is not mandatory, state persistence is still supported**
- ▶ **Support for multiple EventBus and EventStore implementations**
- ▶ **Integrates nicely with Spring**
- ▶ **supports complex business transactions (“sagas” in DDD parlance)**

Things I'm not quite sure about

- ▶ **requires some familiarity with DDD terms (you should at least know what an Aggregate is)**
- ▶ **Axon doesn't try to hide CQRS from developers – whether this good or bad depends on experience and knowledge about CQRS**

That's all I have.
Feel free to ask me anything!

@owolf

innoQ