Doctrine

To Use Or Not To Use

Benjamin Eberlei

September 18, 2015

Me

- Doctrine Project Team-Lead
- blogging `www.whitewashing.de`
- tweeting `@beberlei`

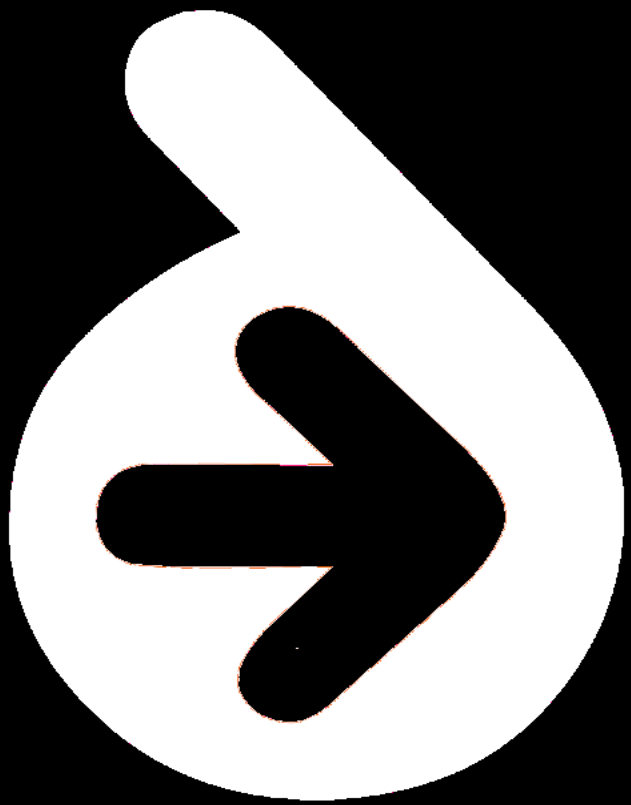
We promote high quality code with trainings and consulting

`http://qafoo.com`

Profiling, Performance Monitoring, Error Tracking for PHP

`https://tideways.io`

Qafoo
passion for software quality



#ormhate

I've often felt that much of the frustration with ORMs is about inflated expectations. (Martin Fowler)

`http://martinfowler.com/bliki/OrmHate.html`

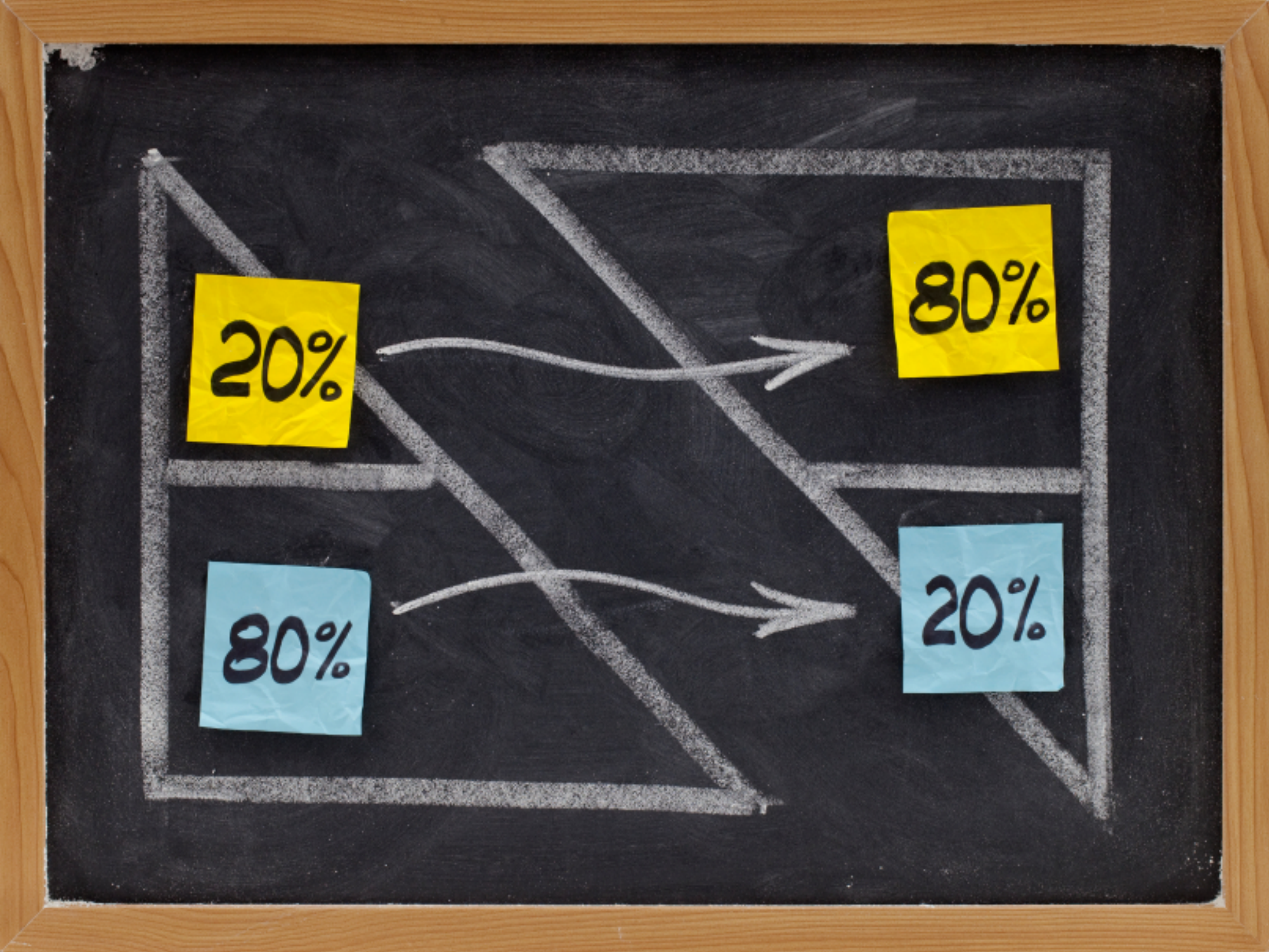
ORM Complexity

Mapping of relational data into memory is

really hard

Vilfredo Pareto (1848-1923)

Licensed under CC BY-SA 3.0 via Commons https://commons.wikimedia.org/wiki/File:Vilfredo_Pareto.jpg


20%
80%
80%
20%

Economics and Doctrine?

We built Doctrine for the 80% use case + `$some`.

Economics and Doctrine?

You can **implement 80%** of your use cases with Doctrine **in a sane way**.

Economics and Doctrine?

Choosing Doctrine does not require you to go all in

What is Doctrine good/bad at?

- Very good: CRUD (and Symfony Validator + Forms)
- Good: ”DDD Lite” with coupling to Doctrine/DB
- Okish: Domain Driven Design (decoupling requires work)
- Bad: (Extremely-) high write throughput
- Bad: Analytics, Statistics, Reporting

Doctrine and Domain-Driven Design

Doctrine enables Domain-Driven Design
to some degree.
BUT
to avoid a meltdown you
must still think in relations.

ORM Building Blocks: The 80%

- CRUD (find, persist, remove, flush)
- Associations
- Lazy Loading
- Repository API
- Doctrine Query Language
- Native Queries

ORM Building Blocks: The $some %

You are at risk of

- Unsolvable Edge Cases
- High Coupling
- Performance Problems
- Complexity

1. Inheritance

1. Inheritance: Lazy Loading

```php
<?php

abstract class Product
{
}

class OrderItem
{
    /**
     * @ORM\ManyToOne(targetEntity="Product")
     */
    private $inventoryItem;
}
```

2. Inheritance

- Lazy Loading does not work on inheritance assocations
- Joined Table Inheritance
  - Slow because of many (hidden) JOINs
  - Moving a field requires moving column between tables

Alternative: The ”nosql” column

```php
<?php

class Product
{
    /**
     * @ORM\Column(type="json_array")
     * @var array
     */
    private $attributes;
}
```

2. Events

2. Events

```php
<?php

$product = $entityManager->find(Product::class, 1234);

$product->setTitle("foo");

$entityManager->flush();
```

2. Events

- Make interaction with UnitOfWork even more complex
- Run on **every** flush or load operation
- Don’t run in context of the domain
- Some of the extensions on top of events are very inefficient

Alternative: Explicit Code

Move behaviour into
entities, services or domain events

3. Batch Processing

3. Batch Processing

```php
<?php

$dql = 'SELECT p FROM Product p';
$query = $entityManager->createQuery($dql);

$it = $query->iterate();
$i = 0;

foreach ($it as $product) {
    $i++;

    $product->setPrice(10);

    if ($i % 20 == 0) {
        $entityManager->flush();
    }
}
```

3. Batch Processing

- ORMs are really bad at this
- Both memory and CPU bottleneck
- Highly coupled code

Alternative: Model Changes

```php
<?php

class UpdatePrice extends ProductChange
{
    public $productId;
    public $price;
}

class UpdateAvailability extends ProductChange
{
    public $productId;
    public $availability;
}
```

Alternative: Model Changes

```php
<?php


$update1 = new UpdatePrice([
    'productId' => 1234,
    'price' => 10,
]);

$update2 = new UpdateAvailability([
    'productId' => 4444,
    'availability' => 120,
]);

$repository->applyChanges([
    $update1,
    $update2
]);
```

4. DQL, Fetch-Joins and Pagination

4. DQL, Fetch-Joins and Pagination

Order	Customer Name	Items	Total	State
#1234	John Doe 53111 Bonn, Germany	4 Acme Super-Widget and 3 more	120 €	Paid
#1234	Petra Musterfrau 10111 Berlin, Germany	2 Acme Super-Widget and one more	27 €	Paid

4. DQL, Fetch-Joins and Pagination

```php
<?php

$dql = 'SELECT o, oi, p, da, sa, c
          FROM Order o
          JOIN o.orderItems oi
          JOIN oi.product p
          JOIN o.deliveryAddress da
          JOIN o.shippingAddress sa
          JOIN o.customer c';

$query = $entityManager->createQuery($query);
$paginator = new Paginator($query);
```

4. DQL, Fetch-Joins and Pagination

- Complex DQL queries with lots of fetches are slow
- Pagination applies some black magic that may not work
- Resulting entities are never exactly what you need to query
- Time-investment to extend DQL is often not worth it

Alternative: View/Read Models

```php
<?php

class OrderListEntry extends Struct
{
    // all the columns
    public $id;
    public $orderNumber;

    // aggregated and computed values
    public $firstItem = array();
    public $totalItems = 0;

    // or re-use value objects
    public $deliveryAddress;

    // ...
}
```

Alternative: View/Read Models

```php
<?php

class OrderRepository extends EntityRepository
{
    /**
     * @return OrderList
     */
    public function listEntries($criteria)
    {
        $connection = $this->getEntityManager()
                           ->getConnection();

        // HERE BE SQL

        return new OrderList();
    }
}
```

How-To build around the ORM in Symfony

The database connection is readily available:

- `$entityManager->getConnection()`
- Service `doctrine.dbal.default_connection`

Doctrine DBAL APIs

```php
<?php

class ProductRepository
{
    private function applyPriceChange($change)
    {
        $connection = $this->getEntityManager()
                           ->getConnection();

        $connection->update([
            'id' => $change->productId,
            'price' => $change->price,
        ]);
    }
}
```

Doctrine DBAL APIs

```php
<?php

class OrderRepository extends EntityRepository
{
    public function listEntries($criteria)
    {
        $connection = $this->getEntityManager()
                           ->getConnection();
        $qb = $connection->createQueryBuilder();
        $qb->from('orders', 'o')
           ->innerJoin(
                'o', 'customers', 'c',
                'o.customer_id = c.id'
            )
        // ...
    }
}
```

Not Using the ORM

You should avoid rolling your own

- Significant time investement
- hard to pull out
- Easier to use existing combined with custom SQL/code

Not Using the ORM

You can use NoSQL

- avoid the mapping complexity
- Not all data is relational

Conclusions

Rule of Thumb #1

Avoid complex mapping by using appropriate in-memory data-structures

Rule of Thumb #2

You still have to think about the database

https://joind.in/14976

Qafoo
passion for software quality

THANK YOU

Rent a quality expert
qafoo.com