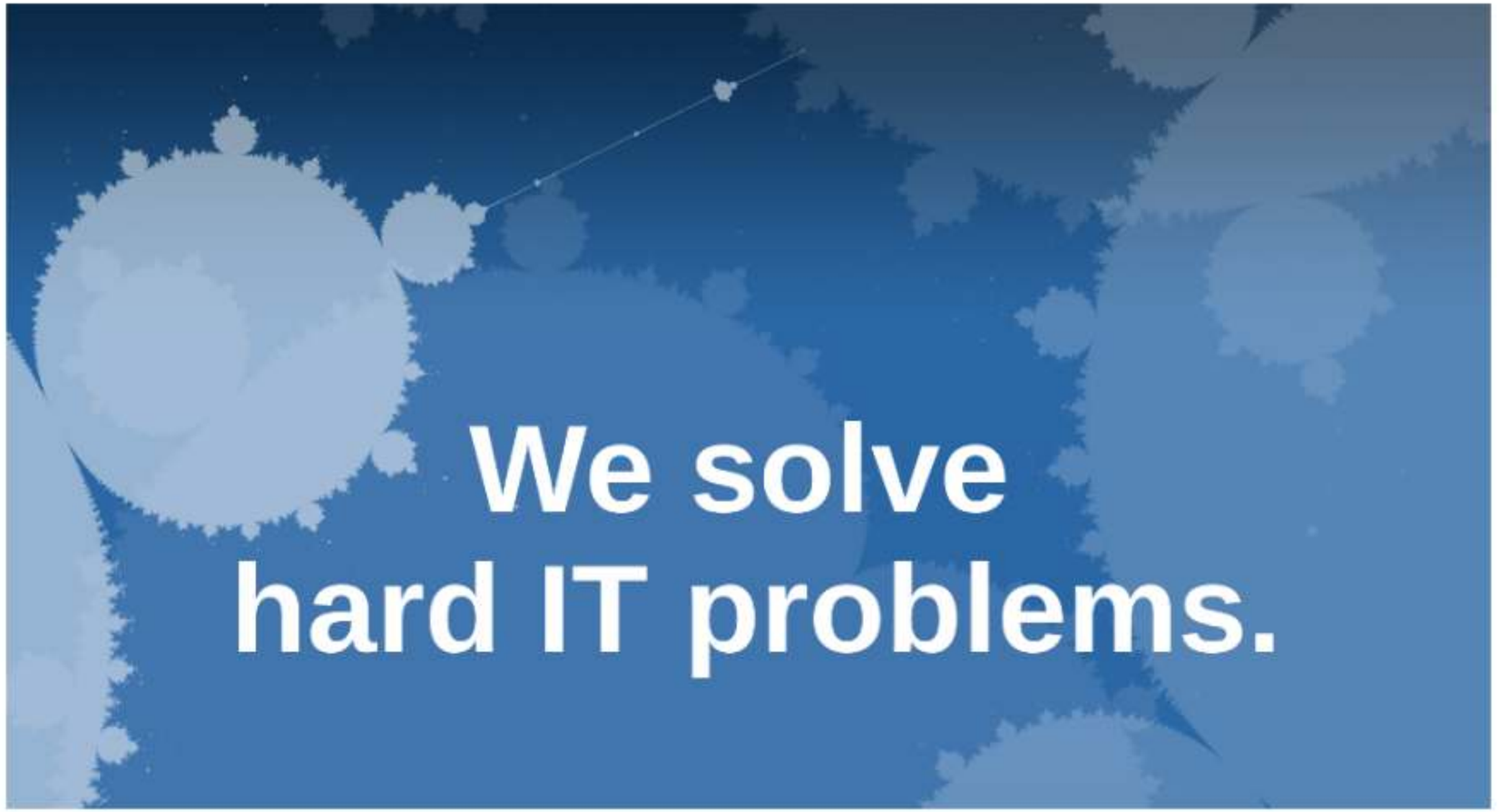
Architecture Refactoring

Moving Towards DDD

Michael Haeuslmann - @michaelhaeu

TNG TECHNOLOGY CONSULTING

Michael Haeuslmann @ TNG TECHNOLOGY CONSULTING

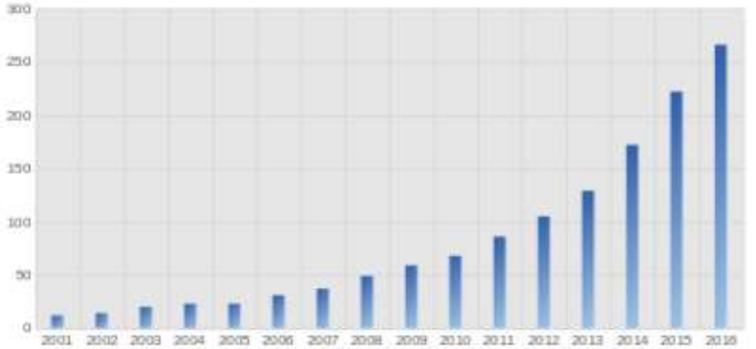


TNG TECHNOLOGY CONSULTING

Facts and Figures

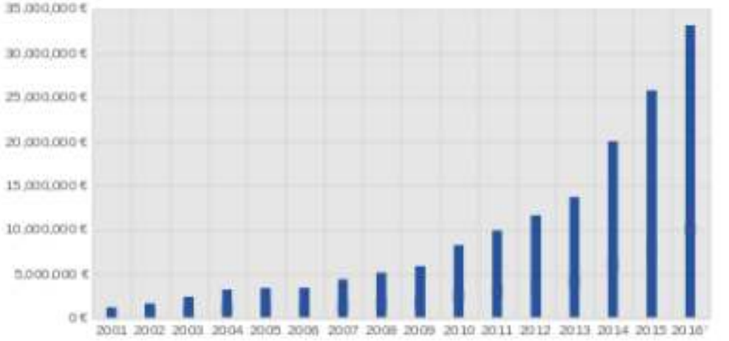
- Foundation in 2001
- Located in Unterföhring near Munich
- 99% University Degree Holder
  60% with a PhD
- 2 Internal Training Days per Month
- Approx. 275 Employees
- 41% with a Degree in Physics
- Fluctuation < 6%
- 1-4 External Conferences per Year

Number of Employees



Note: The number of employees is stated as of the end of December.

Sales Growth



Refactoring Architecture is hard!

Refactoring Architecture is hard!

Just rewrite!

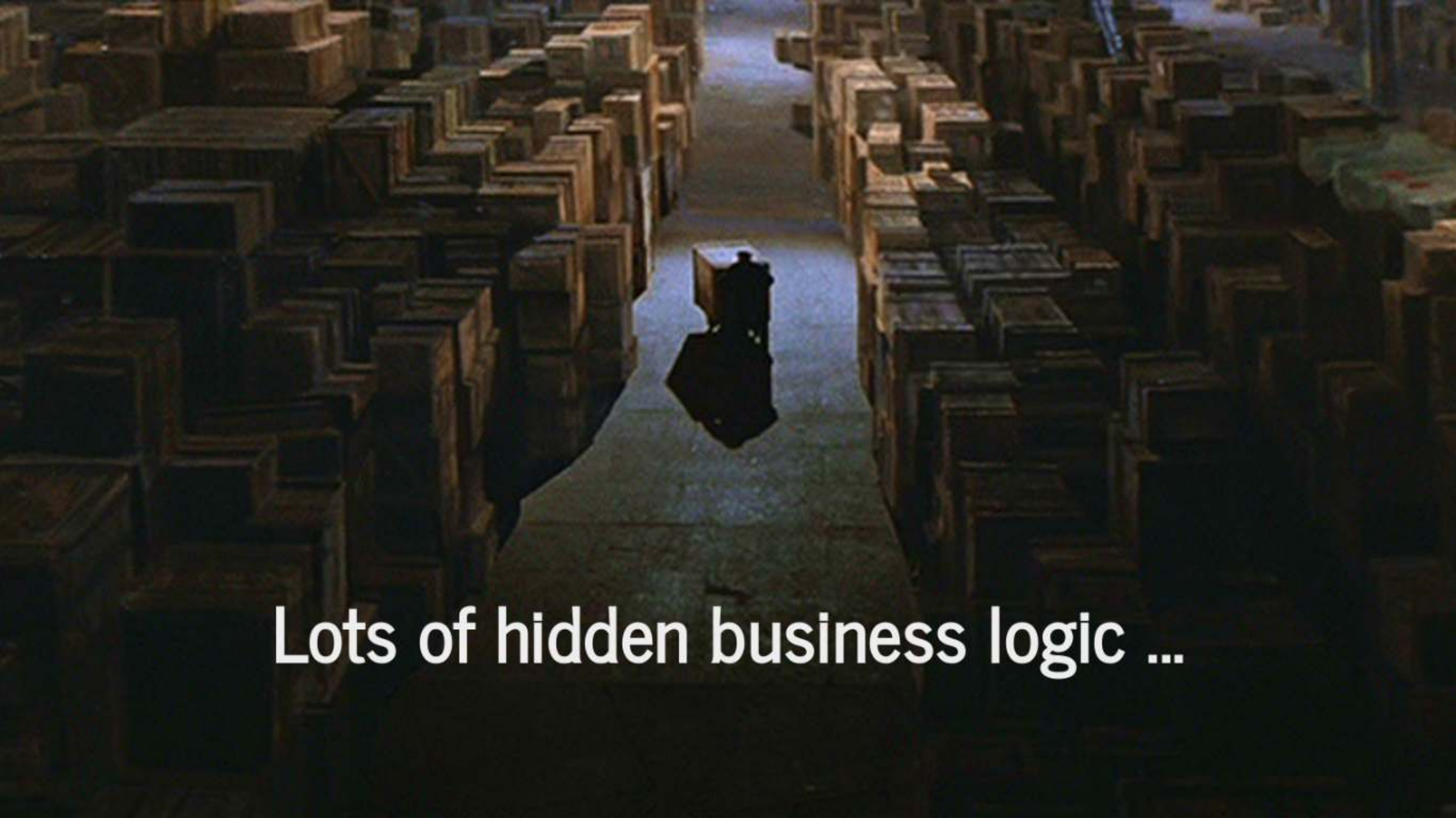

Dubai 27 years ago

Case Study: Vodafone Kabeldeutschland


Rewrite
Just
FOR
YOU


Rev te


Communication
problems ...


Lots of hidden business logic ...


Big bang integration ...


We need a plan!

What do we want?

Architecture which ...

- supports the use cases and operation of the system
- supports the maintenance of the system
- supports the development of the system
- supports the deployment of the system

Good architecture is ...

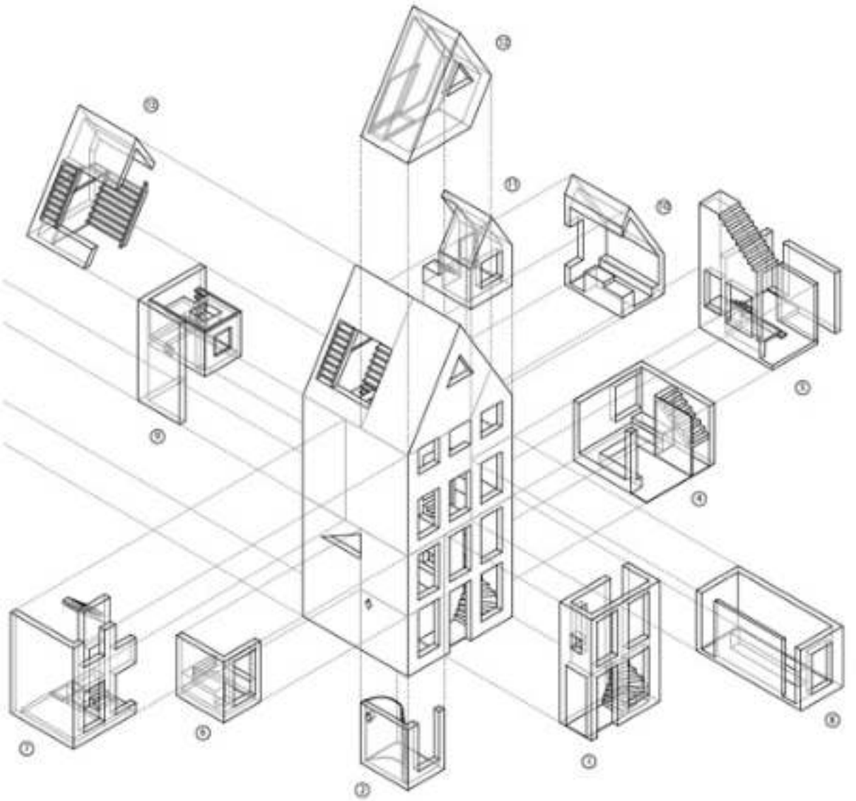
- ... about intent not about methods/tools
- ... allows major decisions to be deferred

Good architecture is ...

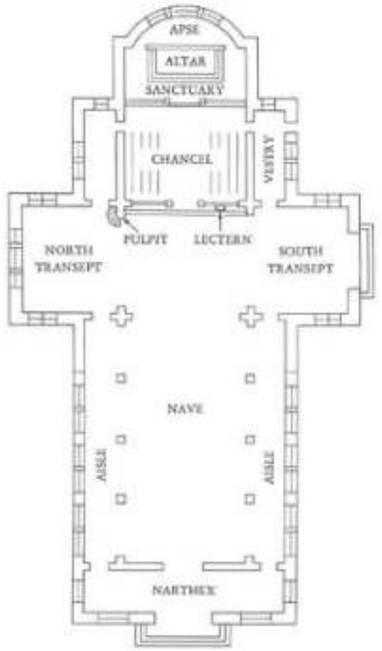
- ... about intent not about methods/tools
- ... allows major decisions to be deferred

- "Architecture, The Lost Years", Robert C. Martin

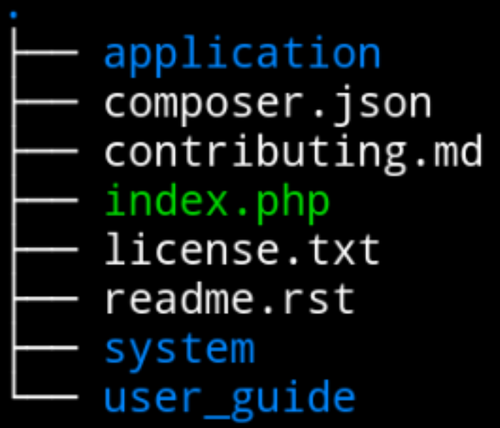
What's the intent?



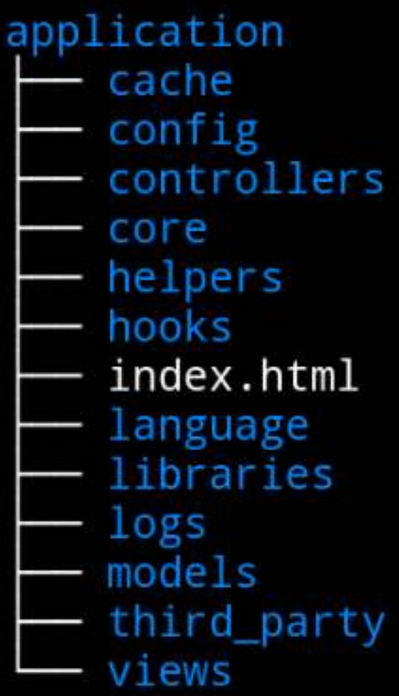
What's the intent?



What's the intent?



What's the intent?



Good architecture is about intent not about methods or tools.

- Robert C. Martin ("Uncle Bob")

Good architecture allows major decisions to be deferred.

- Robert C. Martin ("Uncle Bob")

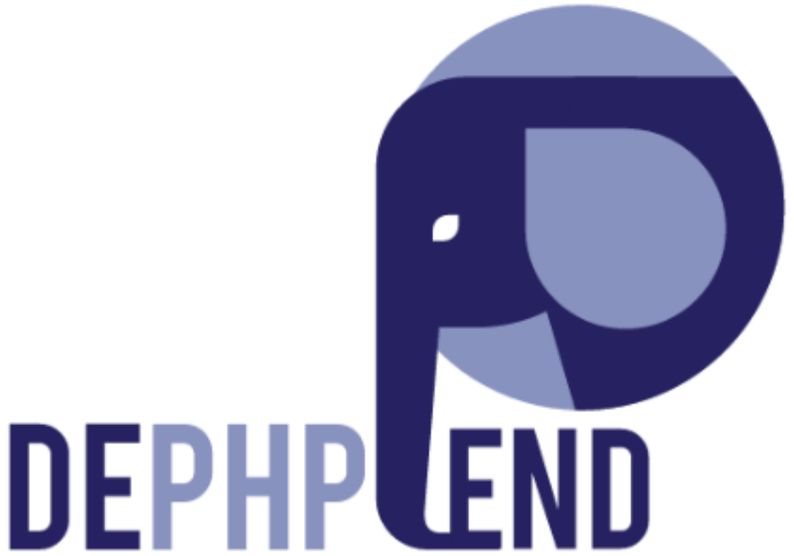
How do we find problems in our architecture?

- read ...
- read ...
- read ...
- read ...
- read ...


STOP


We need some tools!

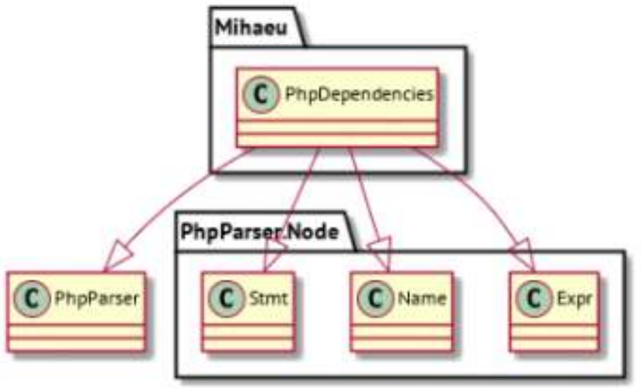
dePHPend

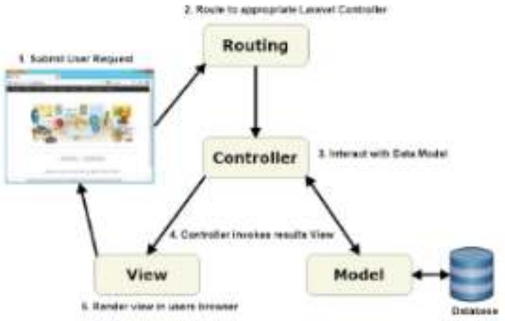


Visualizations & Assertions

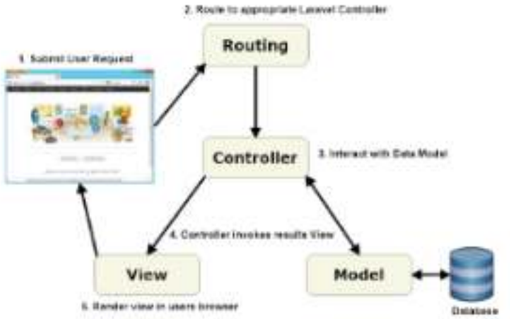

```
λ wget http://phar.dephpend.com/dephpend.phar -O ~/bin/dephpend
λ dephpend --help
      _     _____  _    _ _____                _
     | |    |  __ \| |  | |  __ \              | |
   __| | ___| |__) | |__| | |__) |__ _ __   __| |
  / _` |/ _ \  ___/|  __  |  ___/ _ \ _ \ / _` |
 | (_| |  __/ |    | |  | | |  |  __/ | | | (_| |
  \__,_|\___|_|    |_|  |_|_|   \___|_| |_|\__,_| version 0.4

 Usage: ...
λ dephpend text ~/workspace/dephpend/src
Mihaeu\PhpDependencies\Util\AbstractMap --> Mihaeu\PhpDependencies\Util\Collection
Mihaeu\PhpDependencies\Util\DI --> Mihaeu\PhpDependencies\Analyser\Analyser
...
```

Visualizations



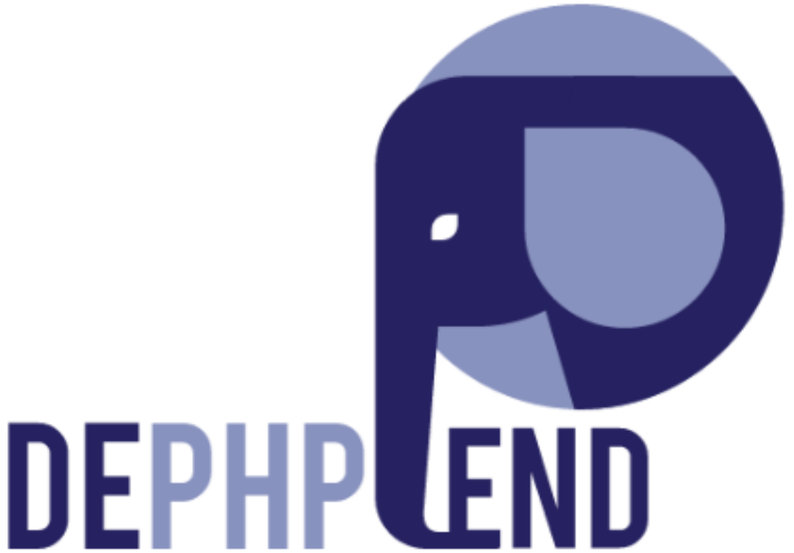

2. Route to appropriate Laravel Controller
Routing
1. Submit User Request
Controller
3. Interact with Data Model
4. Controller invokes results View
View
Model
Database
5. Render view in users browser

DEPHPEND



DEPHPEND

```php
<?php

$cmd = shell_exec('dephpend text src --no-classes');
$constraints = [
    'Model.* --> .*View',
    'View.*  --> .*Model',
];
$regex = '/('.implode(')|(', $constraints).')/x';

if (preg_match($regex, $cmd)) {
    echo 'Architecture violation'.PHP_EOL;
    exit(1);
}
```



`https://github.com/mihaeu/dephpend`

Whatever you end up using:

Test your architecture!

Metrics

- number of dependencies, dependents
- package size
- abstractness of code

pmd and PhpInspectionsEa to detect other architectural smells

- gives you hints along the way
- too many dependencies in a class
- inheritance hierarchy too deep
- static calls

https://phpmd.org/

https://github.com/kalessil/phpinspectionsea

PhpStorm for refactorings and renaming

- no vim is not an alternative
- no neither is emacs
- and don't get me started on Netbeans

Refactoring Browser by Qafoo

For JavaScript

- Visualizations: grep for require/import + dot (graphviz)
- Assertions: grep for require/import + scripts
- pmd for JavaScript
- WebStorm or VSCode

One more thing: Functional Tests

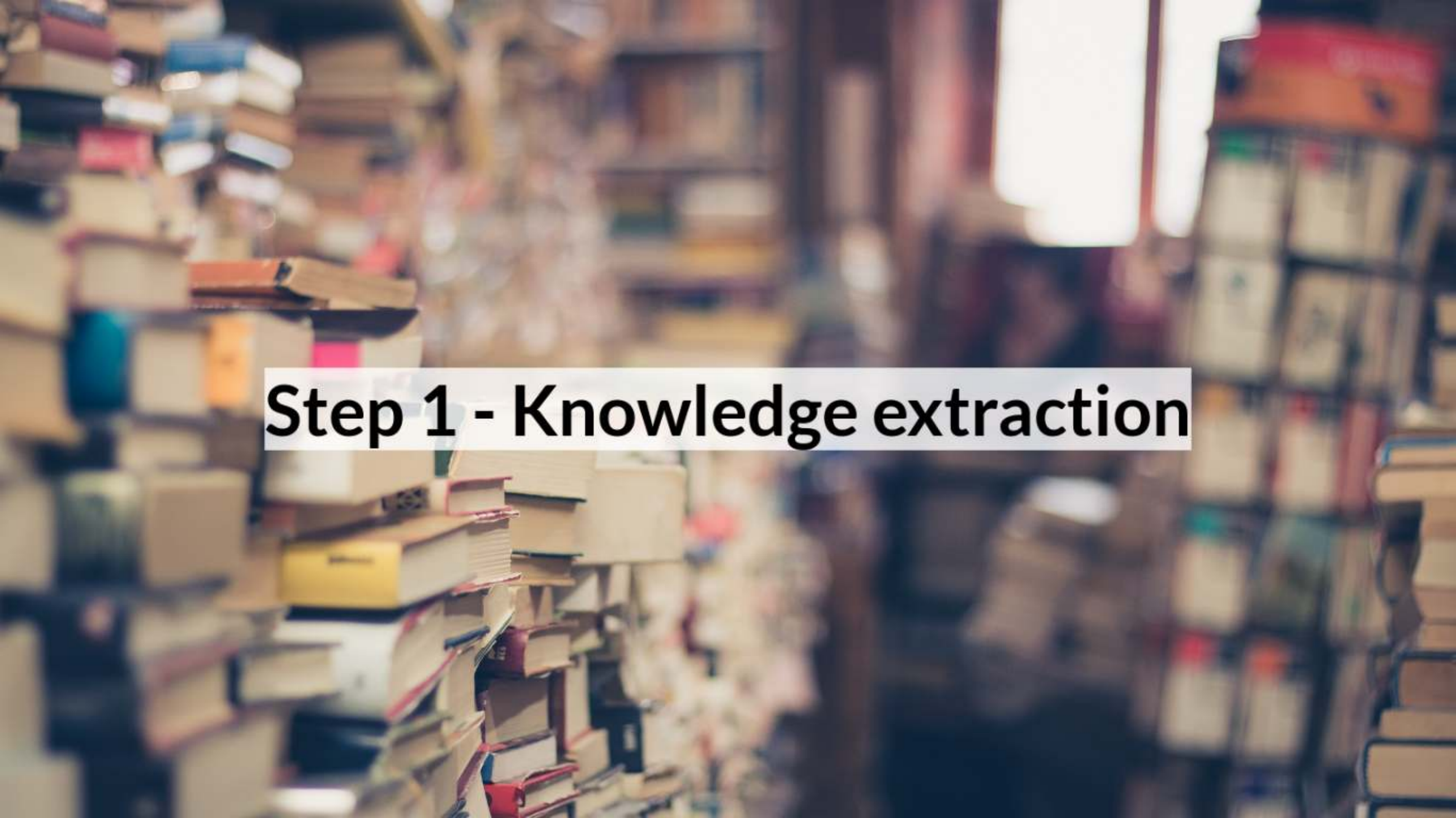
⇒ refactoring should not break existing functionality

⇒ functional tests are more resilient towards architectural changes

Behat

Nightwatch.js


Let's begin



Step 1 - Knowledge extraction

What's this?

```
/** @var string */
private $address;
```

- what does it contain?
- how is it validated?
- where is it validated?
- is the validation always the same?
- when is it populated?
- who uses it?

Or that?

```
/** @var array */
private $address;
```

- not much better
- most questions remain unanswered

Is this okay?

```
class AddressStruct
{
    public $street;
    public $zipCode;
    public $city;
}
```

- tiny step in the right direction
- helps with usages
- state is unknown and mutable
- still difficult to refactor

Immutable Domain Objects

```
class Address {
    private $street;
    private $zipCode;
    private $city;

    public function __construct(
        Street $street, ZipCode $zipCode, City $city
    ) {
        // ...
    }

    public function street() : Street { return $this->street; }
}
```


L
M

Immutable Domain Objects

- is the Address valid?
- ⇒ as long as it exists it's valid
- unused attributes?
- ⇒ the IDE will know
- adding/removing properties?
- ⇛ all related unit tests will break (which is great)
- what if a value changes?
- ⇒ they can't

Immutable Domain Objects

```
class ZipCode
{
    private $zipCode;

    public function __construct(int $zipCode)
    {
        $this->ensureValidGermanZipCode($zipCode);

        $this->zipCode = $zipCode;
    }

    private function ensureValidGermanZipCode(int $zipCode) : void
    {
        if ($zipCode < 10000 || $zipCode > 99999) {
            throw new InvalidZipCodeException($zipCode);
        }
    }
}
```

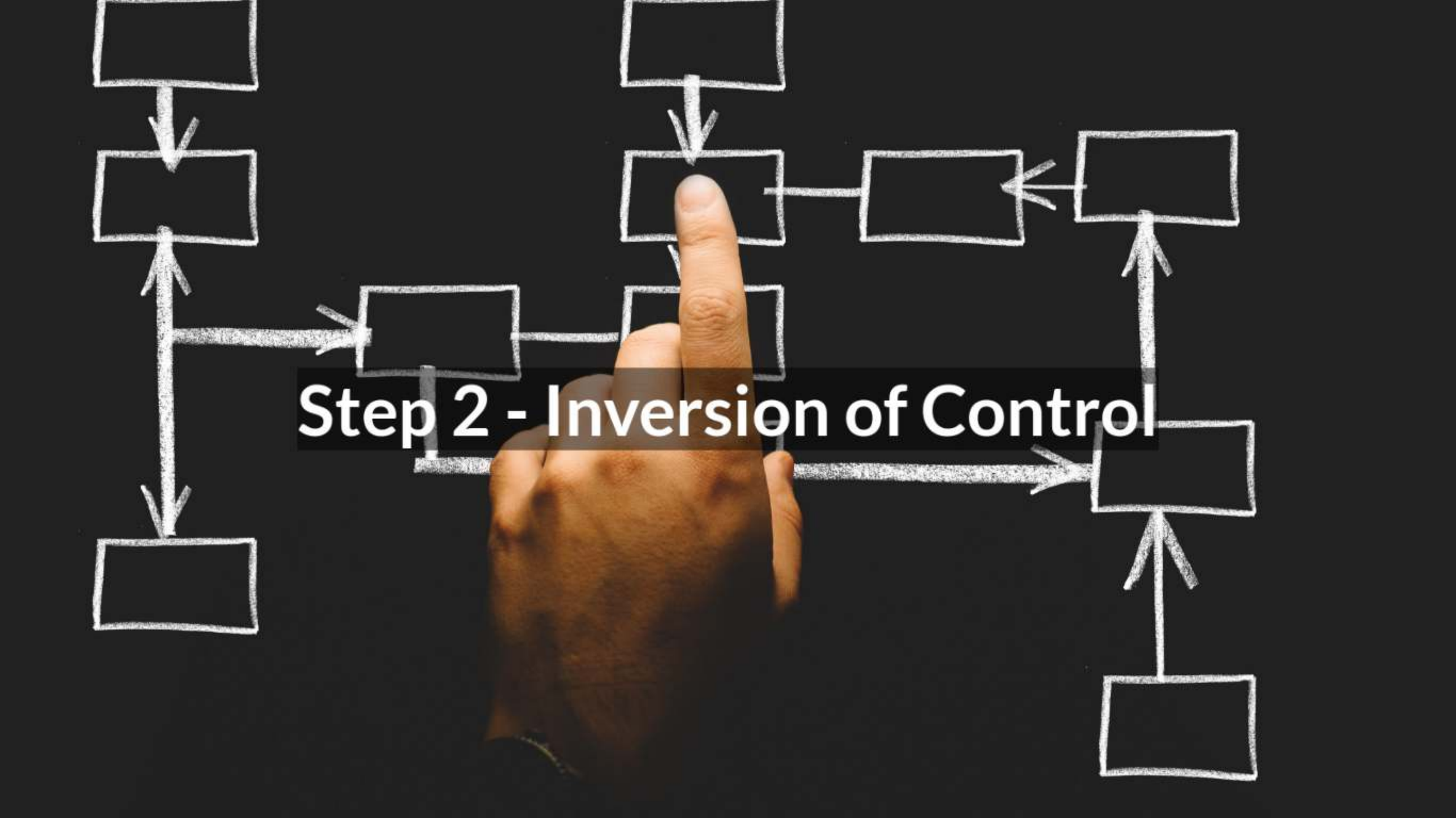
Summary: Immutable Domain Objects

- not an architectural refactoring, but important to reason about the business logic
- looks easy, but very painful process
- you'll find many inconsistencies along the way
- Knowledge gain: "who talks to whom about what?"
- Myth: arrays are more performant (*)

Summary: Immutable Domain Objects

- not an architectural refactoring, but important to reason about the business logic
- looks easy, but very painful process
- you'll find many inconsistencies along the way
- Knowledge gain: "who talks to whom about what?"
- Myth: arrays are more performant (*)

(*) PHP data structures and the impact of PHP 7 on them


Step 2 - Inversion of Control

What do I depend on?

```php
class X
{
  public function __construct($dependencyContainer)
  {
    $this->dependencyContainer = $dependencyContainer;
  }

  // ... a few hundred lines later

  public function someFunction()
  {
    $userRepository = $this->dependencyContainer->get('UserRepository');
    $users = $userRepository->getAll();
  }
}
```

What do I depend on?

```
class X
{
  // ... a few hundred lines later

  public function someFunction()
  {
    $users = DB::connection('foo')->select(/** ... */);
  }
}
```

What do I depend on?

```
class X
{
  // ... a few hundred lines later

  public function someFunction()
  {
    $users = DB::connection('foo')->select(/** ... */);
  }
}
```

Don't make me think about stupid things!

Dependency Injection

- always use constructor dependency injection
- never inject the dependency container itself
- domain objects are "newables"
- factories are fine
- use a lib or roll your own, the less magic the better

Common Problems

Side Effects in the Constructor

```php
function __construct($db) {
    $this->db = $db;
    $this->init(); // oh oh
}
```

- Option 1: lazy instantiation using e.g. factories
- Option 2: Extract side effects (temporal coupling 😳)

Common Problems

Static Usage

```
class UserHelper
{
    public static function isAuthorized() { /** ... */ }
}
```

Common Problems

Static Usage

```
class UserHelper
{
    public static function isAuthorized() { /** ... */ }
}
```

Wrap it!

Common Problems

Static Usage

```
class UserHelper
{
    public static function isAuthorized() { /** ... */ }
}
```

Wrap it!

```
class UserHelperWrapper
{
    public function isAuthorized() {
        return UserHelper::isAuthorized();
    }
}
```

Common Problems

Framework uses magic

- common problem with most frameworks
- magic instantiation of controllers

⇒ if possible write a custom dispatcher

⇒ strangler pattern

⇒ some dependency containers support legacy frameworks


Step 3 - Show Intent
What?!

Be explicit!

- move away from the old-school MVC folder structure
- your folder structure should reflect your architecture
- your architecture should reflect your intent (domain)

Be explicit!

- move away from the old-school MVC folder structure
- your folder structure should reflect your architecture
- your architecture should reflect your intent (domain)

IT is hard, don't make me think about stupid things!

Cohesion

Measure of how well modules fits together.

- Coincidental cohesion
- Logical association
- Temporal cohesion
- Sequential cohesion
- Functional cohesion

Cohesion

Measure of how well modules fits together.

- Coincidental cohesion
- Logical association
- Temporal cohesion
- Sequential cohesion
- Functional cohesion

In other words: things that are highly cohesive should be grouped together

Step 4 - Abstract the delivery mechanism

Abstract the framework/database/delivery mechanism

- the web is a delivery mechanism
- your framework is a delivery mechanism
- delivery mechanisms have nothing to do with your core application
- the source of data should not matter (relational, document, service call, cache, ...)

Common Problems and Questions

- using active record? ⇒ hide them behind a `Repository`
- do your controllers know what happens to the data?
- do your controllers know where the data comes from?
- how expensive would it be to implement event sourcing?
- the source of data should not matter
  (relational, document, service call, cache, ...)

Common Problems and Questions

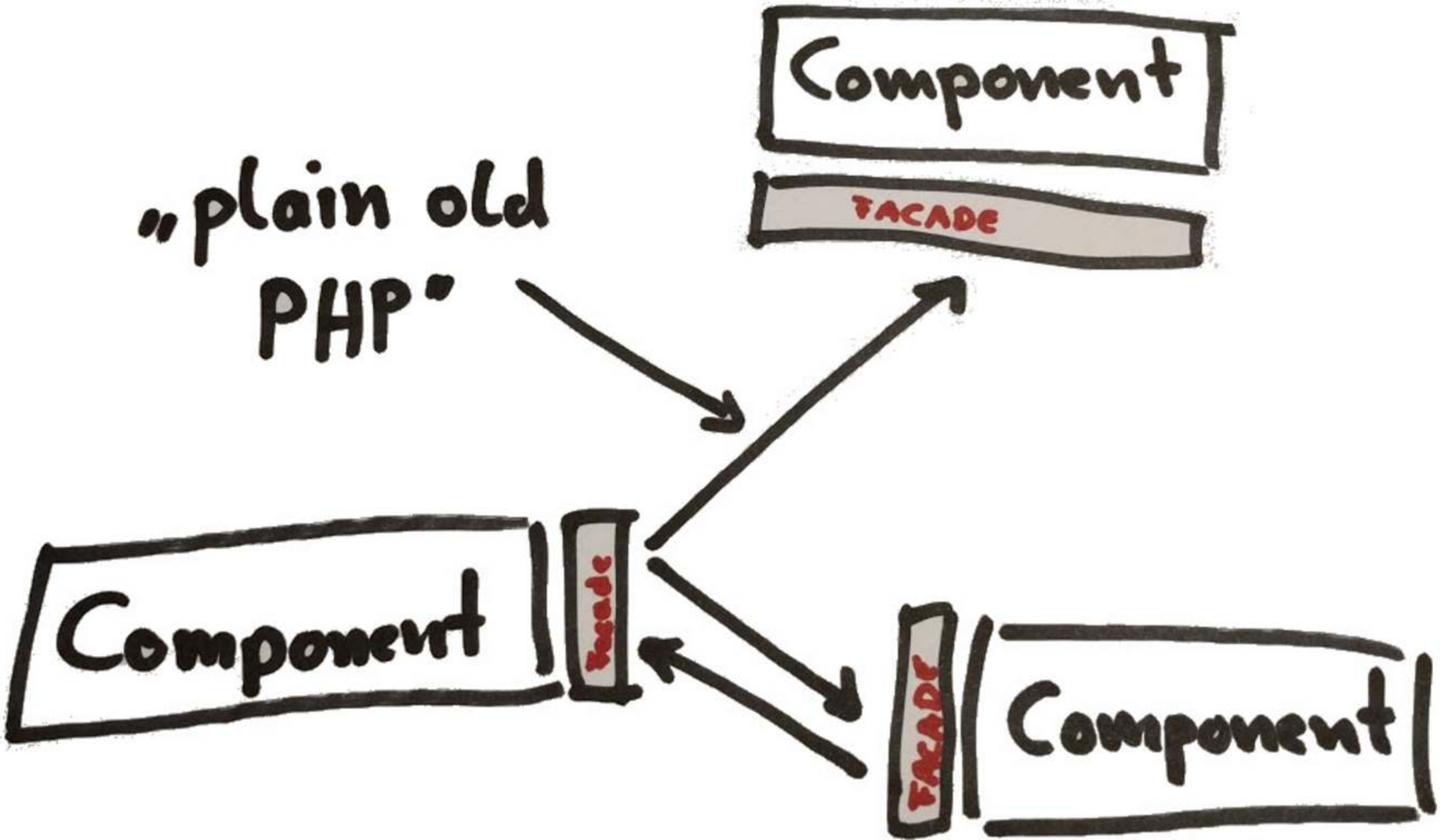
- using active record? ⇒ hide them behind a `Repository`
- do your controllers know what happens to the data?
- do your controllers know where the data comes from?
- how expensive would it be to implement event sourcing?
- the source of data should not matter
  (relational, document, service call, cache, ...)

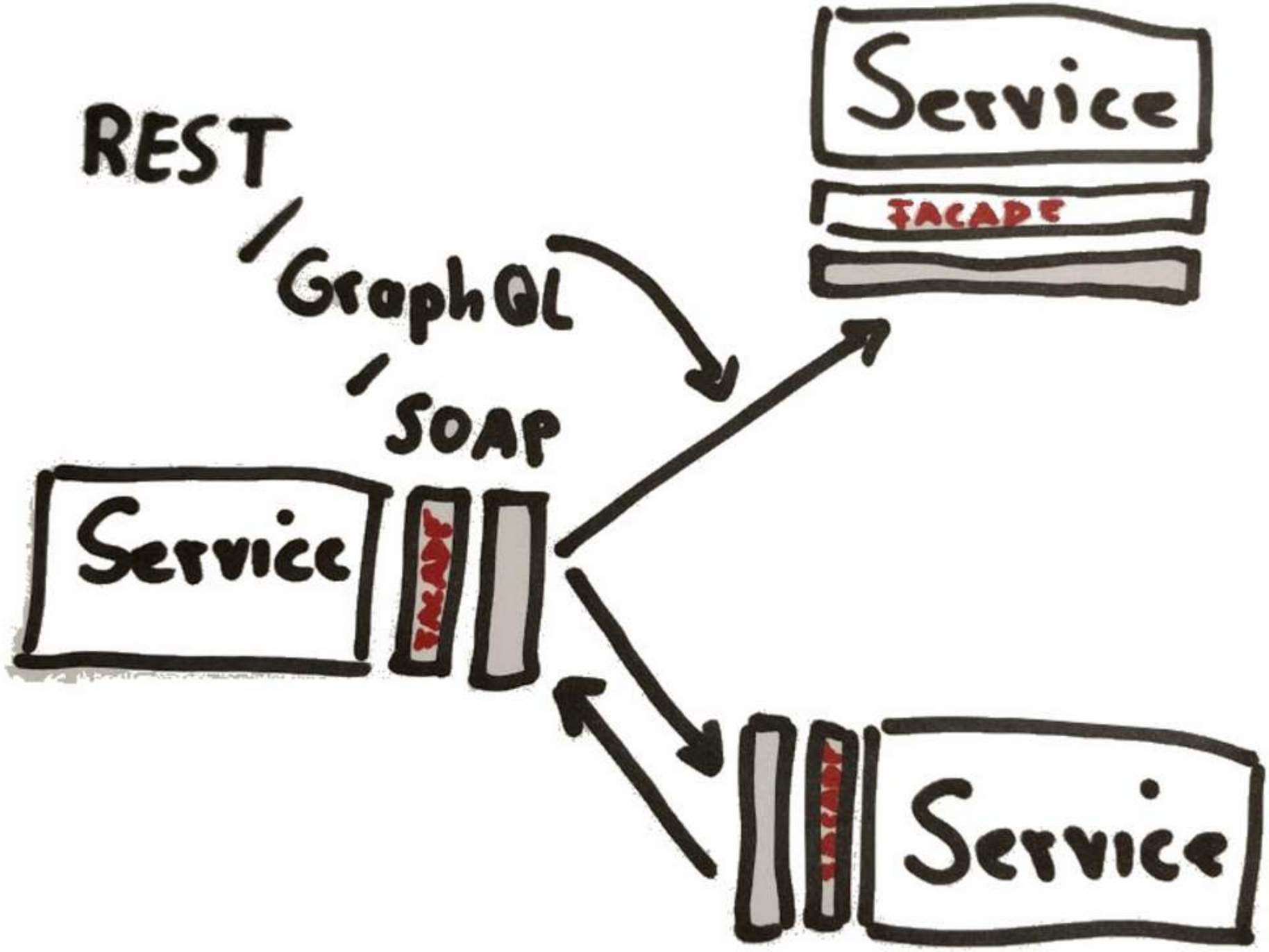
Could you move your core code out of your "web repository"?


Let the fun begin!

Microservices

- you already have your components with clear interfaces
- those interfaces have to be mapped to REST/GraphQL/SOAP requests/responses
- that's about it
  (given the previous steps were successful)


Component
FACADE
„plain old PHP"
Component
FACADE
FACADE
Component


REST
/
GraphQL
/
SOAP
Service
FACADE
Service
FACADE
FACADE
Service

Fallacies of distributed computing

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Domain Driven Design (DDD)

- we already built domain objects
- aggregates should be easy to identify between the different components
  it's what your facades should be using anyways
- layered architecture has been ensured using constraints
- service access is encapsulated
- storage access is encapsulated
- however: introducing events is not free

Domain Driven Design (DDD)

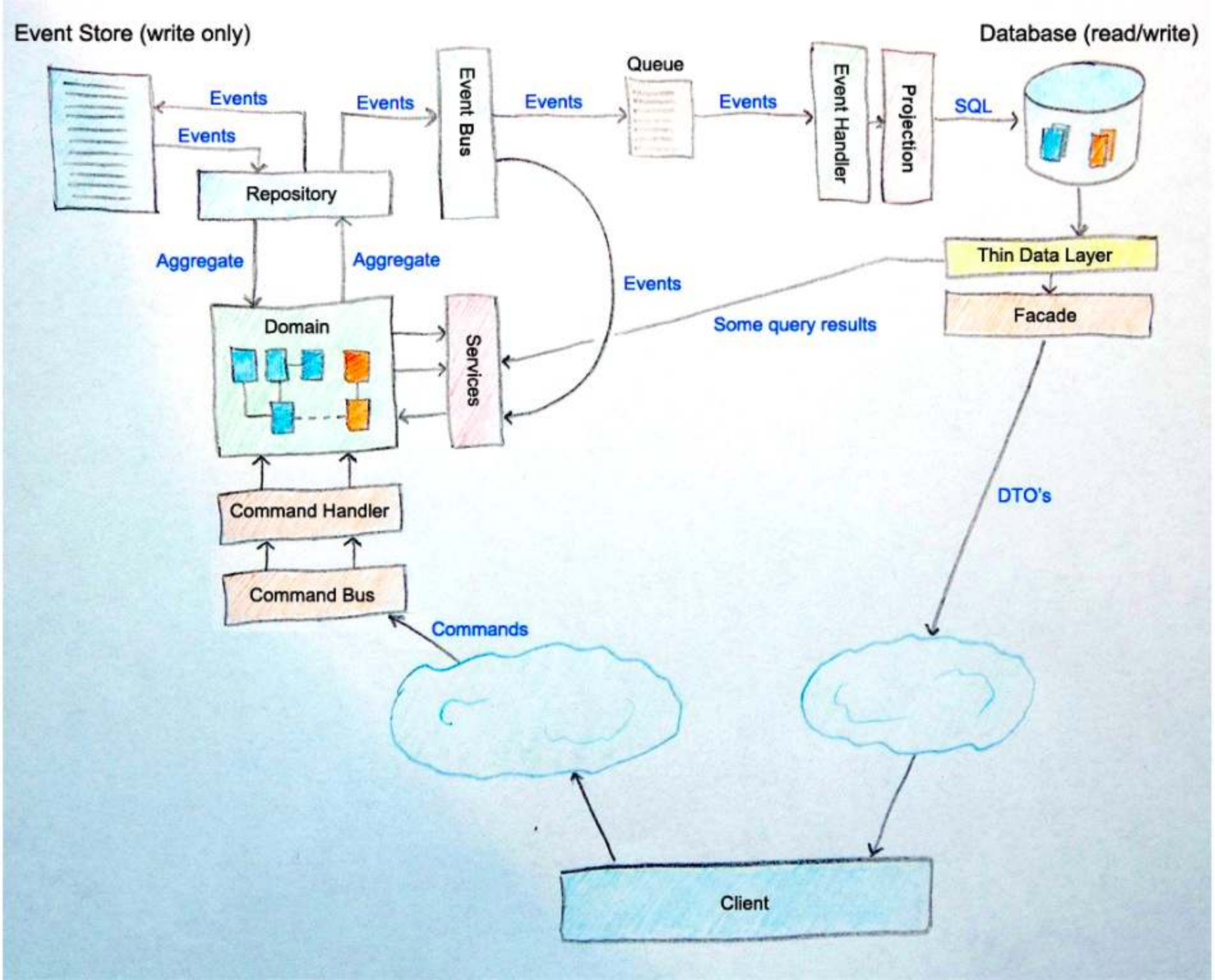
- we already built domain objects
- aggregates should be easy to identify between the different components
  it's what your facades should be using anyways
- layered architecture has been ensured using constraints
- service access is encapsulated
- storage access is encapsulated
- however: introducing events is not free

Optional Step 5 - CQRS

Command Query Responsibility Segregation

Event Sourcing

- the data source is abstract (e.g. repository)
- the rest of the application doesn't care about the source
- send requests from a client through the command bus
- application deals with commands and talks to repository
- repository maps what happened to events

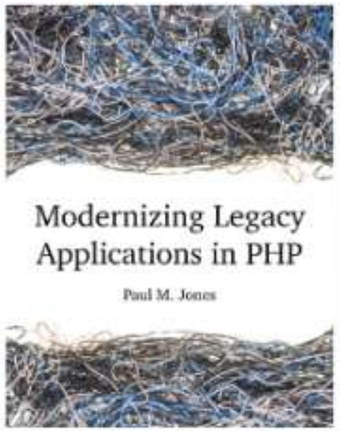

Event Store (write only)
Database (read/write)
Events
Events
Events
Events
Event Bus
Events
Queue
Events
Event Handler
Projection
SQL
Repository
Aggregate
Aggregate
Thin Data Layer
Events
Domain
Services
Some query results
Facade
DTO's
Command Handler
Command Bus
Commands
Client

Summary

- architecture refactorings are costly and risky
- make sure your goals are measurable
- define and test architecture constraints

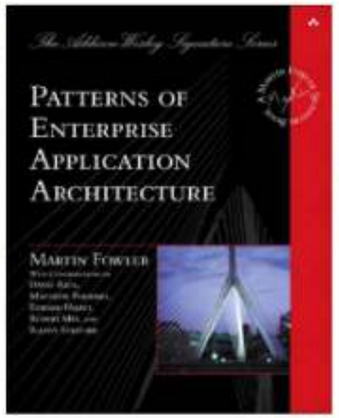
1. Knowledge Extraction (Domain modeling)
2. Inversion of Control
3. Abstract the delivery mechanism
4. Show Intent

Reading Recommendations

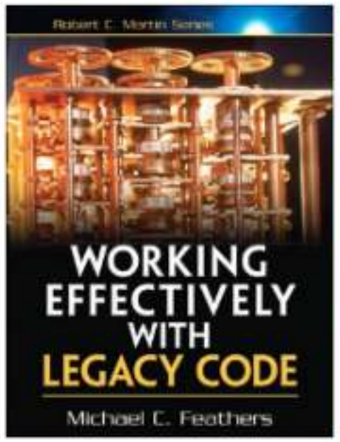


Modernizing Legacy Applications in PHP
(Paul M. Jones)

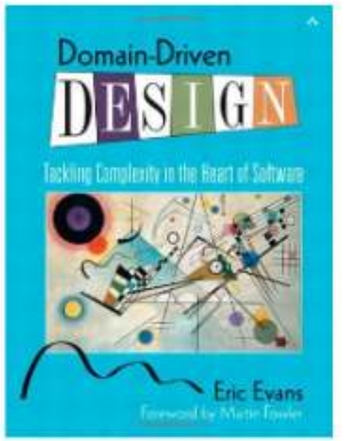
Clean Architecture
(Robert C. Martin)



Patterns of Enterprise Application Architecture
(Martin Fowler)



Working Effectively with Legacy Code
(Michael C. Feathers)



Domain Driven Design
(Eric Evans)

Thank you!

Michael Haeuslmann - @michaelhaeu

TNG TECHNOLOGY CONSULTING

Credits

```
Stock photos from pexel.com
Indiana Jones http://raven.theraider.net/showthread.php?t=8100&pag
Big Bang https://www.youtube.com/watch?v=hDcWqidxvz4
CQRS à la Greg Young - Mark Nijhof
Cohesion https://courses.cs.washington.edu/courses/cse403/96sp/cou
```