Advanced OO Patterns

PHP UK Conference 2011

Tobias Schlitt <toby@qafoo.com>

February 25, 2011

Outline

About me

- Degree in computer sience
- More than 10 years of professional PHP enthusiast
- Zeta Components
- PHPUnit

About me

- Degree in computer sience
- More than 10 years of professional PHP

About me

- Degree in computer sience
- More than 10 years of professional PHP
- Open source enthusiast
  - Apache Zeta Components
  - Arbit
  - PHPUnit
  - ...

About me

- Degree in computer sience
- More than 10 years of professional PHP
- Open source enthusiast
  - Apache Zeta Components
  - Arbit
  - PHPUnit
  - ...

Co-Founder of
Qafoo GmbH
http://qafoo.com

About me

- Degree in computer sience
- More than 10 years of professional PHP
- Open source enthusiast
  - Apache Zeta Components
  - Arbit
  - PHPUnit
  - ...

Co-Founder of
Qafoo GmbH
`http://qafoo.com`

We help people to produce high quality PHP code.

Disclaimer

- This talk cannot be in depth
- This talk shall inspire you
- This talk will not show UML diagrams
- This talk will show you quite some code
- This talk can seriously harm your coding habbits

Disclaimer

- This talk cannot be in depth
- This talk shall inspire you

Disclaimer

- This talk cannot be in depth
- This talk shall inspire you
- This talk will not show UML diagrams

Disclaimer

- This talk cannot be in depth
- This talk shall inspire you
- This talk will not show UML diagrams
- This talk will show you quite some code

can seriously harm your coding habbits

Disclaimer

- This talk cannot be in depth
- This talk shall inspire you
- This talk will not show UML diagrams
- This talk will show you quite some code
- This talk can seriously harm your coding habbits

Patterns are . . .

- . . . universal solutions.

Patterns are . . .

- ~~. . . universal solutions.~~

Patterns are . . .

- ~~. . . universal solutions.~~
- . . . good habbits.

Patterns are . . .

- ~~. . . universal solutions.~~
- ~~. . . good habbits.~~

Patterns are . . .

- ~~. . . universal solutions.~~
- ~~. . . good habbits.~~
- . . . coding templates.

Patterns are . . .

- ~~. . . universal solutions.~~
- ~~. . . good habbits.~~
- ~~. . . coding templates.~~

Patterns are . . .

- ~~. . . universal solutions.~~
- ~~. . . good habbits.~~
- ~~. . . coding templates.~~

. . . names for proven ideas how a certain class of problems can be solved.

Patterns are not . . .

- . . . appliable to every problem.
- . . . directly transferable to code.
- . . . always the best solution.

Pattern classification

- Creational
- Structural
- Behavioural
- Architectural

Well known patterns

Signal / Observer

Iterator

Visitor

Adapter

Singleton

Factory

Well known patterns

Signal / Observer

Iterator

Visitor

Adapter

~~**Singleton**~~

Factory

Outline

Goals of good OO design

- Modular
- Flexible
- Reusable

(www.unclebob.com/ArticleS.UncleBob.PrinciplesOfOod)

Goals of good OO design

- Modular
- Flexible

Goals of good OO design

- Modular
- Flexible
- Reusable

Goals of good OO design

- Modular
- Flexible
- Reusable
- Testable

(www.unclebob.com/ArticleS.UncleBob.PrinciplesOfOod)

Goals of good OO design

- Modular
- Flexible
- Reusable
- Testable
- S.O.L.I.D.

  (`http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`)

What's the problem here?

```
1  <?php
2
3  class MessageDispatcher
4  {
5      public function __construct()
6      {
7          $this->messengers[] = new
               JabberMessenger();
8          $this->messengers[] = new
               MailMessenger();
9      }
10 }
11
12 class MailMessenger implements Messenger
13 {
14     public function sendMessage( $text )
15     {
16         myLogger::getInstance()->log( $text
               );
17         $this->sendMail( /*...*/ );
18     }
19 }
```

What's the problem here?

```
1  <?php
2
3  class MessageDispatcher
4  {
5      public function __construct()
6      {
7          $this->messengers[] = new
               JabberMessenger();
8          $this->messengers[] = new
               MailMessenger();
9      }
10 }
11
12 class MailMessenger implements Messenger
13 {
14     public function sendMessage( $text )
15     {
16         myLogger::getInstance()->log( $text
               );
17         $this->sendMail( /*...*/ );
18     }
19 }
```

- Inflexible
- Not re...
- Hardly testable

What's the problem here?

```
1  <?php
2
3  class MessageDispatcher
4  {
5      public function __construct()
6      {
7          $this->messengers[] = new
               JabberMessenger();
8          $this->messengers[] = new
               MailMessenger();
9      }
10 }
11
12 class MailMessenger implements Messenger
13 {
14     public function sendMessage( $text )
15     {
16         myLogger::getInstance()->log( $text
               );
17         $this->sendMail( /*...*/ );
18     }
19 }
```

- Inflexible
- Not reusable
- Hardly testable

What's the problem here?

```
1  <?php
2
3  class MessageDispatcher
4  {
5      public function __construct()
6      {
7          $this->messengers[] = new
               JabberMessenger();
8          $this->messengers[] = new
               MailMessenger();
9      }
10 }
11
12 class MailMessenger implements Messenger
13 {
14     public function sendMessage( $text )
15     {
16         myLogger::getInstance()->log( $text
               );
17         $this->sendMail( /*...*/ );
18     }
19 }
```

- Inflexible
- Not reusable
- Hardly testable

Injecting dependencies

```php
<?php

$messenger = new MessageDispatcher(
    array(
        new JabberMessenger( 'jabber.example.org', 'user', 'pass' ),
        new MailMessenger(
            new MailSmtpTransport( 'mail.example.org', 'user', 'pass' ),
            $logger = new Logger(
                new LoggingDispatcher(
                    array(
                        new SyslogLogger(),
                        new FileSystemLogger( 'log/errors.log' )
                    )
                )
            )
        )
    ),
    $logger
);
```

Injecting dependencies

```php
<?php

$messenger = new MessageDispatcher(
    array(
        new JabberMessenger( 'jabber.example.org', 'user', 'pass' ),
        new MailMessenger(
            new MailSmtpTransport( 'mail.example.org', 'user', 'pass' ),
            $logger = new Logger(
                new LoggingDispatcher(
                    array(
                        new SyslogLogger(),
                        new FileSystemLogger( 'log/errors.log' )
                    )
                )
            )
        )
    ),
    $logger
);
```

Injecting dependencies

```php
<?php

$messenger = new MessageDispatcher(
    array(
        new JabberMessenger( 'jabber.example.org', 'user', 'pass' ),
        new MailMessenger(
            new MailSmtpTransport( 'mail.example.org', 'user', 'pass' ),
            $logger = new Logger(
                new LoggingDispatcher(
                    array(
                        new SyslogLogger(),
                        new FileSystemLogger( 'log/errors.log' )
                    )
                )
            )
        )
    ),
    $logger
);
```

Injecting dependencies

```
1  <?php
2
3  $messenger = new MessageDispatcher(
4      array(
5          new JabberMessenger( 'jabber.example.org', 'user', 'pass' ),
6          new MailMessenger(
7              new MailSmtpTransport( 'mail.example.org', 'user', 'pass' ),
8              $logger = new Logger(
9                  new LoggingDispatcher(
10                     array(
11                         new SyslogLogger(),
12                         new FileSystemLogger( 'log/errors.log' )
13                     )
14                 )
15             )
16         )
17     ),
18     $logger
19 );
```

Injecting dependencies

```php
<?php

$messenger = new MessageDispatcher(
    array(
        new JabberMessenger( 'jabber.example.org', 'user', 'pass' ),
        new MailMessenger(
            new MailSmtpTransport( 'mail.example.org', 'user', 'pass' ),
            $logger = new Logger(
                new LoggingDispatcher(
                    array(
                        new SyslogLogger(),
                        new FileSystemLogger( 'log/errors.log' )
                    )
                )
            )
        )
    ),
    $logger
);
```

Injecting dependencies

```
1  <?php
2
3  $messenger = new MessageDispatcher(
4      array(
5          new JabberMessenger( 'jabber.example.org', 'user', 'pass' ),
6          new MailMessenger(
7              new MailSmtpTransport( 'mail.example.org', 'user', 'pass' ),
8              $logger = new Logger(
9                  new LoggingDispatcher(
10                     array(
11                         new SyslogLogger(),
12                         new FileSystemLogger( 'log/errors.log' )
13                     )
14                 )
15             )
16         )
17     ),
18     $logger
19 );
```

Dependency injection

- Pros
  - Flexibility
  - Reusability
  - Modularity
  - Testability
- Cons
  - Complex object graphs
  - Long parameter lists
  - Object ballast

Dependency injection container

```php
<?php

class DependencyInjectionContainer
{
    protected $messageDispatcher;
    protected $logger;

    public function __construct(
        MessageDispatcher $messageDispatcher,
        Logger $logger
    )
    {
        $this->messageDispatcher;
        $this->logger;
    }

    public function getMessageDispatcher()
    {
        return $this->messageDispatcher;
    }

    public function getLogger()
    {
        return $this->logger;
    }
}
```

Outline

Motivation

- Building the full object graph is
  - Complex
  - Time consuming
  - Memory consuming

Motivation

- Building the full object graph is
  - Complex
  - Time consuming
  - Memory consuming
- Do you use all objects in every request?

Motivation

- Building the full object graph is
  - Complex
  - Time consuming
  - Memory consuming
- Do you use all objects in every request?
- Idea: Initialize object when needed for the first time

Simple lazy initialization

```php
<?php

class DatabaseInitializer
{
    protected $dsn;

    protected $db;

    public function __construct( $dsn )
    {
        $this->dsn = $dsn;
    }

    public function getDatabase()
    {
        if ( $this->db === null )
        {
            $this->db = new Database( $this->dsn );
        }
        return $this->db;
    }
}
```

Simple lazy initialization

```php
<?php

class DatabaseInitializer
{
    protected $dsn;

    protected $db;

    public function __construct( $dsn )
    {
        $this->dsn = $dsn;
    }

    public function getDatabase()
    {
        if ( $this->db === null )
        {
            $this->db = new Database( $this->dsn );
        }
        return $this->db;
    }
}
```

Simple lazy initialization

```php
<?php

class DatabaseInitializer
{
    protected $dsn;

    protected $db;

    public function __construct( $dsn )
    {
        $this->dsn = $dsn;
    }

    public function getDatabase()
    {
        if ( $this->db === null )
        {
            $this->db = new Database( $this->dsn );
        }
        return $this->db;
    }
}
```

Combining DIC and lazy initialization

- Inject everything
- Initialize only when needed
- Resolve dependencies automatically

The Arbit DIC

- Exemplary for a high-end DIC
  - Shared objects (initialized once)
  - Closures for lazy initialization
  - Inherent dependency resolution

The Arbit DIC

- Exemplary for a high-end DIC
  - Shared objects (initialized once)
  - Closures for lazy initialization
  - Inherent dependency resolution
- Base class for custom DICs

The Arbit DIC

- Exemplary for a high-end DIC
  - Shared objects (initialized once)
  - Closures for lazy initialization
  - Inherent dependency resolution
- Base class for custom DICs
- Heavy use of interceptors

ntation details,
http://bit.ly/arbitDIC

The Arbit DIC

- Exemplary for a high-end DIC
  - Shared objects (initialized once)
  - Closures for lazy initialization
  - Inherent dependency resolution
- Base class for custom DICs
- Heavy use of interceptors
- No implementation details,
  see here: `http://bit.ly/arbitDIC`

Using the Arbit DIC

```
3  class arbitEnvironmentDIC extends arbitDependencyInjectionContainer
4  {


13     public function initialize()
14     {


38     }
39 }
```

Using the Arbit DIC

```
3  class arbitEnvironmentDIC extends arbitDependencyInjectionContainer
4  {

13     public function initialize()
14     {

15         $this->cache = function( $dic )
16         {
17             return new arbitFilesystemCache( $dic->request->controller );
18         };

38     }
39 }
```

Using the Arbit DIC

```
3  class arbitEnvironmentDIC extends arbitDependencyInjectionContainer
4  {

13     public function initialize()
14     {

20         $this->messenger = function( $dic )
21         {
22             return new arbitMessenger( array(
23                 'email' => $dic->mailMessenger,
24             ) );
25         };

38     }
39 }
```

Using the Arbit DIC

```
3   class arbitEnvironmentDIC extends arbitDependencyInjectionContainer
4   {

13      public function initialize()
14      {

27          $this->mailMessenger = function( $dic )
28          {
29              return new arbitMailMessenger(
30                  $dic->mailTransport,
31                  $dic->configuration->main,
32                  $dic->configuration->project( $dic->request->controller ),
33                  $dic->views->decorator
34              );
35          };

38      }
39  }
```

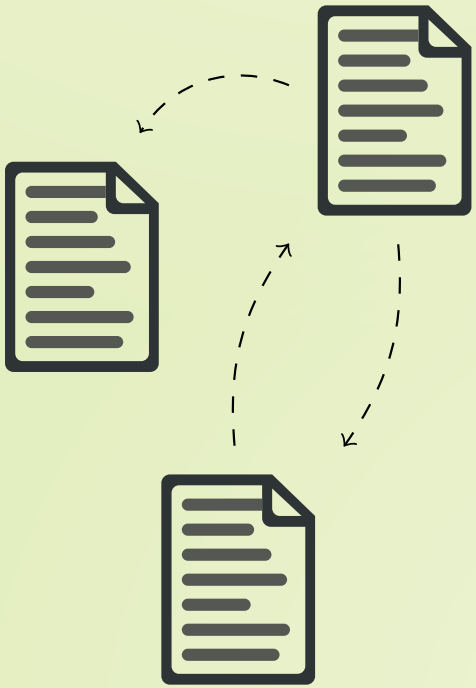
More about DIC

- Pimple - A small PHP 5.3 dependency injection container
  `https://github.com/fabpot/Pimple`
- Bucket - Basic di-container for php
  `https://github.com/troelskn/bucket`
- Symfony Dependency Injection `http://components.symfony-project.org/dependency-injection/`
- **Martin Fowler on Dependency Injection**
  `http://martinfowler.com/articles/injection.html`

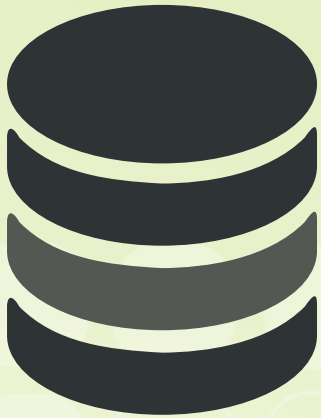
Outline

The situation

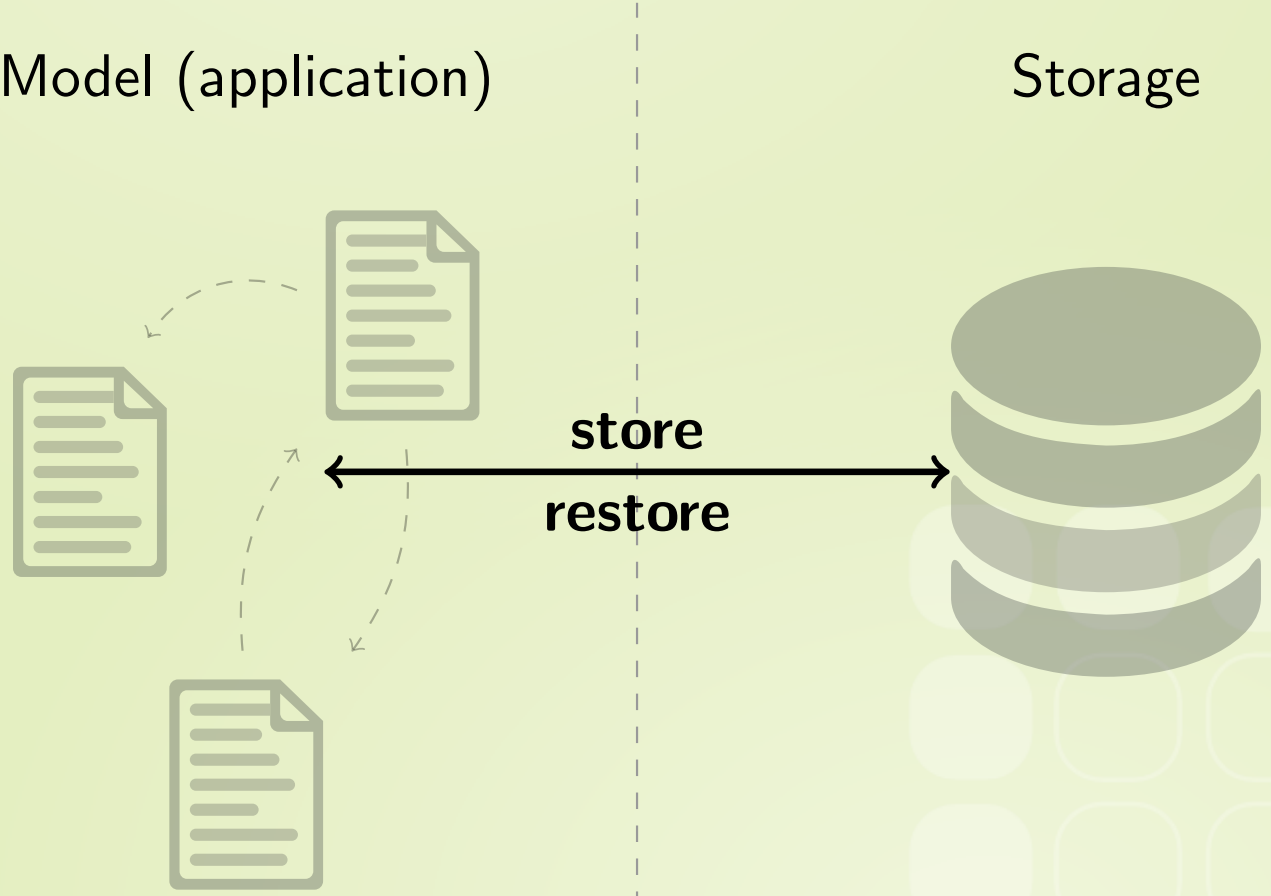
Model (application)



Storage



The situation



Challenges

- Model and storage structure differ
  - Object relational impedance mismatch
- Different access approaches
  - Query language differences
  - Backend might change
  - Could even be mixed

Challenges

- Model and storage structure differ
  - Object relational impedance mismatch
- Different access approaches
  - Query language differences
  - Backend might change
  - Could even be mixed

Challenges

- Model and storage structure differ
  - Object relational impedance mismatch
- Different access approaches

Challenges

- Model and storage structure differ
  - Object relational impedance mismatch
- Different access approaches
  - Query language differences

Challenges

- Model and storage structure differ
  - Object relational impedance mismatch
- Different access approaches
  - Query language differences
- Storage back end might change

Challenges

- Model and storage structure differ
  - Object relational impedance mismatch
- Different access approaches
  - Query language differences
- Storage back end might change
- Back ends could even be mixed

Challenges

- Model and storage structure differ
  - Object relational impedance mismatch
- Different access approaches
  - Query language differences
- Storage back end might change
- Back ends could even be mixed
- ...

Active Record

```php
<?php

class Invoice extends ActiveRecord
{
    protected $id;
    protected $positions;
    protected $vat;
    // ...

    public function calculateValue()
    { /* business logic */ }
}

class ActiveRecord
{
    public function insert()
    {  /* ... */  }
    public function update()
    { /* ... */ }
}
```

Active Record

```php
<?php

class Invoice extends ActiveRecord
{
    protected $id;
    protected $positions;
    protected $vat;
    // ...

    public function calculateValue()
    { /* business logic */ }
}

class ActiveRecord
{
    public function insert()
    {  /* ... */  }
    public function update()
    { /* ... */ }
}
```

Active Record

```php
<?php

class Invoice extends ActiveRecord
{
    protected $id;
    protected $positions;
    protected $vat;
    // ...

    public function calculateValue()
    { /* business logic */ }
}

class ActiveRecord
{
    public function insert()
    {  /* ... */  }
    public function update()
    { /* ... */ }
}
```

Active Record

```php
<?php

class Invoice extends ActiveRecord
{
    protected $id;
    protected $positions;
    protected $vat;
    // ...

    public function calculateValue()
    { /* business logic */ }
}

class ActiveRecord
{
    public function insert()
    {  /* ... */  }
    public function update()
    { /* ... */ }
}
```

Active Record

- Combines storage and business logic
- Commonly storage logic in base class
- Model class corresponds to database record

Evaluation

Pros

- Very easy to use
- Very few code to write

Cons

- Model structure = storage structure
- Classes become complex
  - Business logic
  - Storage logic
- Broken object semantics
- Changes ripple over
- Hard to exchange storage logic
- Really hard to test!

Evaluation

Pros

- Very easy to use
- Very few code to write

Cons

- Model structure = storage structure
- Classes become complex
  - Business logic
  - Storage logic
- Broken object semantics
- Changes ripple over
- Hard to exchange storage logic
- Really hard to test!

Evaluation

Pros

- Very easy to use
- Very few code to write

Cons

- Model structure = storage structure
- Classes become complex
  - Business logic
  - Storage logic
- Broken object semantics
- Changes ripple over
- Hard to exchange storage logic
- Really hard to test!

Evaluation

Pros

- Very easy to use
- Very few code to write

Cons

- Model structure = storage structure
- Classes become complex
  - Business logic
  - Storage logic
- Broken object semantics
- Changes ripple over
- Hard to exchange storage logic
- Really hard to test!

Evaluation

Pros

- Very easy to use
- Very few code to write

Cons

- Model structure = storage structure
- Classes become complex
  - Business logic
  - Storage logic
- Broken object semantics
- Changes ripple over
- Hard to exchange storage logic
- Really hard to test!

Evaluation

Pros

- Very easy to use
- Very few code to write

Cons

- Model structure = storage structure
- Classes become complex
  - Business logic
  - Storage logic
- Broken object semantics
- Changes ripple over
- Hard to exchange storage logic
- Really hard to test!

Evaluation

Pros

- Very easy to use
- Very few code to write

Cons

- Model structure = storage structure
- Classes become complex
  - Business logic
  - Storage logic
- Broken object semantics
- Changes ripple over
- Hard to exchange storage logic
- Really hard to test!

Evaluation

Pros

- Very easy to use
- Very few code to write

Cons

- Model structure = storage structure
- Classes become complex
  - Business logic
  - Storage logic
- Broken object semantics
- Changes ripple over
- Hard to exchange storage logic
- Really hard to test!

Lesson learned . . .

De-couple business and storage logic!

Row Data Gateway

```php
<?php

class Invoice
{
    protected $data;
    // ...

    public function __construct( InvoiceGateway $data )
    { $this->data = $data; }

    public function calculateValue()
    { /* business logic */ }
}

class InvoiceGateway
{
    protected $id;
    protected $positions;
    protected $vat;

    public function insert()
    { /* ... */ }
    public function update()
    { /* ... */ }
}
```

Row Data Gateway

```php
<?php

class Invoice
{
    protected $data;
    // ...

    public function __construct( InvoiceGateway $data )
    { $this->data = $data; }

    public function calculateValue()
    { /* business logic */ }
}

class InvoiceGateway
{
    protected $id;
    protected $positions;
    protected $vat;

    public function insert()
    { /* ... */ }
    public function update()
    { /* ... */ }
}
```

Row Data Gateway

```php
<?php

class Invoice
{
    protected $data;
    // ...

    public function __construct( InvoiceGateway $data )
    { $this->data = $data; }

    public function calculateValue()
    { /* business logic */ }
}

class InvoiceGateway
{
    protected $id;
    protected $positions;
    protected $vat;

    public function insert()
    { /* ... */ }
    public function update()
    { /* ... */ }
}
```

Row Data Gateway

```php
<?php

class Invoice
{
    protected $data;
    // ...

    public function __construct( InvoiceGateway $data )
    { $this->data = $data; }

    public function calculateValue()
    { /* business logic */ }
}

class InvoiceGateway
{
    protected $id;
    protected $positions;
    protected $vat;

    public function insert()
    { /* ... */ }
    public function update()
    { /* ... */ }
}
```

Row Data Gateway

- Decouples storage from business logic
- Still convenient OO interface for storage
- Still some objects coupled to storage

(Gateway is a somewhat similar approach)

Row Data Gateway

- Decouples storage from business logic
- Still convenient OO interface for storage
- Still some objects coupled to storage
- (Table Data Gateway is a somewhat similar approach)

Evaluation

Pros

- Easy to use
- Few code to write
  - Gateways can be generated

Cons

- Some objects model DB structure
- Changes still ripple over
- Still hard to exchange storage logic

Evaluation

Pros

- Easy to use
- Few code to write
  - Gateways can be generated
- Easier to test

Cons

- Some objects model DB structure
- Changes still ripple over
- Still hard to exchange storage logic

Evaluation

Pros

- Easy to use
- Few code to write
  - Gateways can be generated
- Easier to test

Cons

- Some objects model DB structure
- Changes still ripple over
- Still hard to exchange storage logic

Evaluation

Pros

- Easy to use
- Few code to write
  - Gateways can be generated
- Easier to test

Cons

- Some objects model DB structure
- Changes still ripple over
- Still hard to exchange storage logic

Evaluation

Pros

- Easy to use
- Few code to write
  - Gateways can be generated
- Easier to test

Cons

- Some objects model DB structure
- Changes still ripple over
- Still hard to exchange storage logic

Data Mapper

```php
<?php

class Invoice
{
    protected $id;
    protected $positions;
    protected $vat;
    // ...

    public function calculateValue()
    { /* business logic */ }
}

interface InvoiceMapper
{
    public function store( Invoice $invoice );
    public function update( Invoice $invoice );
}

class DbInvoiceMapper implements InvoiceMapper
{
    // ...
}

?>
```

Data Mapper

```php
<?php

class Invoice
{
    protected $id;
    protected $positions;
    protected $vat;
    // ...

    public function calculateValue()
    { /* business logic */ }
}

interface InvoiceMapper
{
    public function store( Invoice $invoice );
    public function update( Invoice $invoice );
}

class DbInvoiceMapper implements InvoiceMapper
{
    // ...
}

?>
```

Data Mapper

```php
<?php

class Invoice
{
    protected $id;
    protected $positions;
    protected $vat;
    // ...

    public function calculateValue()
    { /* business logic */ }
}

interface InvoiceMapper
{
    public function store( Invoice $invoice );
    public function update( Invoice $invoice );
}

class DbInvoiceMapper implements InvoiceMapper
{
    // ...
}

?>
```

Data Mapper

```php
<?php

class Invoice
{
    protected $id;
    protected $positions;
    protected $vat;
    // ...

    public function calculateValue()
    { /* business logic */ }
}

interface InvoiceMapper
{
    public function store( Invoice $invoice );
    public function update( Invoice $invoice );
}

class DbInvoiceMapper implements InvoiceMapper
{
    // ...
}

?>
```

Data Mapper

- Decouple storage from model
- No OO modelling of DB structure
- Model does not even know a database exists

Evaluation

Pros

- Complete decoupling

Cons

Evaluation

Pros

- Complete decoupling
- Model is not aware of storage

Cons

Evaluation

Pros

- Complete decoupling
- Model is not aware of storage
- Clean storage interface

Cons

Evaluation

Pros

- Complete decoupling
- Model is not aware of storage
- Clean storage interface
  - Implement different storages!

Cons

Evaluation

Pros

- Complete decoupling
- Model is not aware of storage
- Clean storage interface
  - Implement different storages!
- DB changes only affect mapping layer
- Model changes only affect mapping layer

Cons

- Quite some code to write
- Mapping can become complex

Evaluation

Pros

- Complete decoupling
- Model is not aware of storage
- Clean storage interface
  - Implement different storages!
- DB changes only affect mapping layer
- Model changes only affect mapping layer
- Nice for testing

Cons

- Queries are hard to write
- Mapping can become complex

Evaluation

Pros

- Complete decoupling
- Model is not aware of storage
- Clean storage interface
  - Implement different storages!
- DB changes only affect mapping layer
- Model changes only affect mapping layer
- Nice for testing

Cons

- Quite some code to write
- Mapping can become complex

Evaluation

Pros

- Complete decoupling
- Model is not aware of storage
- Clean storage interface
  - Implement different storages!
- DB changes only affect mapping layer
- Model changes only affect mapping layer
- Nice for testing

Cons

- Quite some code to write
- Mapping can become complex

Attribution

The ideas behind the storage patterns are from

Patterns of Enterprise Application Architecture

by Martin Fowler

Highy recommended!

Conclusion

- Patterns are not the holy grail!
- They assign names to good ideas
- They help you to talk about concepts
- They inspire you

Conclusion

- Patterns are not the holy grail!
- They assign names to good ideas
- They help you to talk about concepts
- They can inspire you

Thanks for listening

Are there any questions left?

Thanks for listening

Please rate this talk at

`http://joind.in/2512`

and / or give me some feedback right now!

Thanks for listening

Please rate this talk at
`http://joind.in/2512`
and / or give me some feedback right now!

Stay in touch

- Tobias Schlitt
- toby@qafoo.com
- @tobySen / @qafoo

Rent a PHP quality expert:
`http://qafoo.com`