Things to consider for testable code

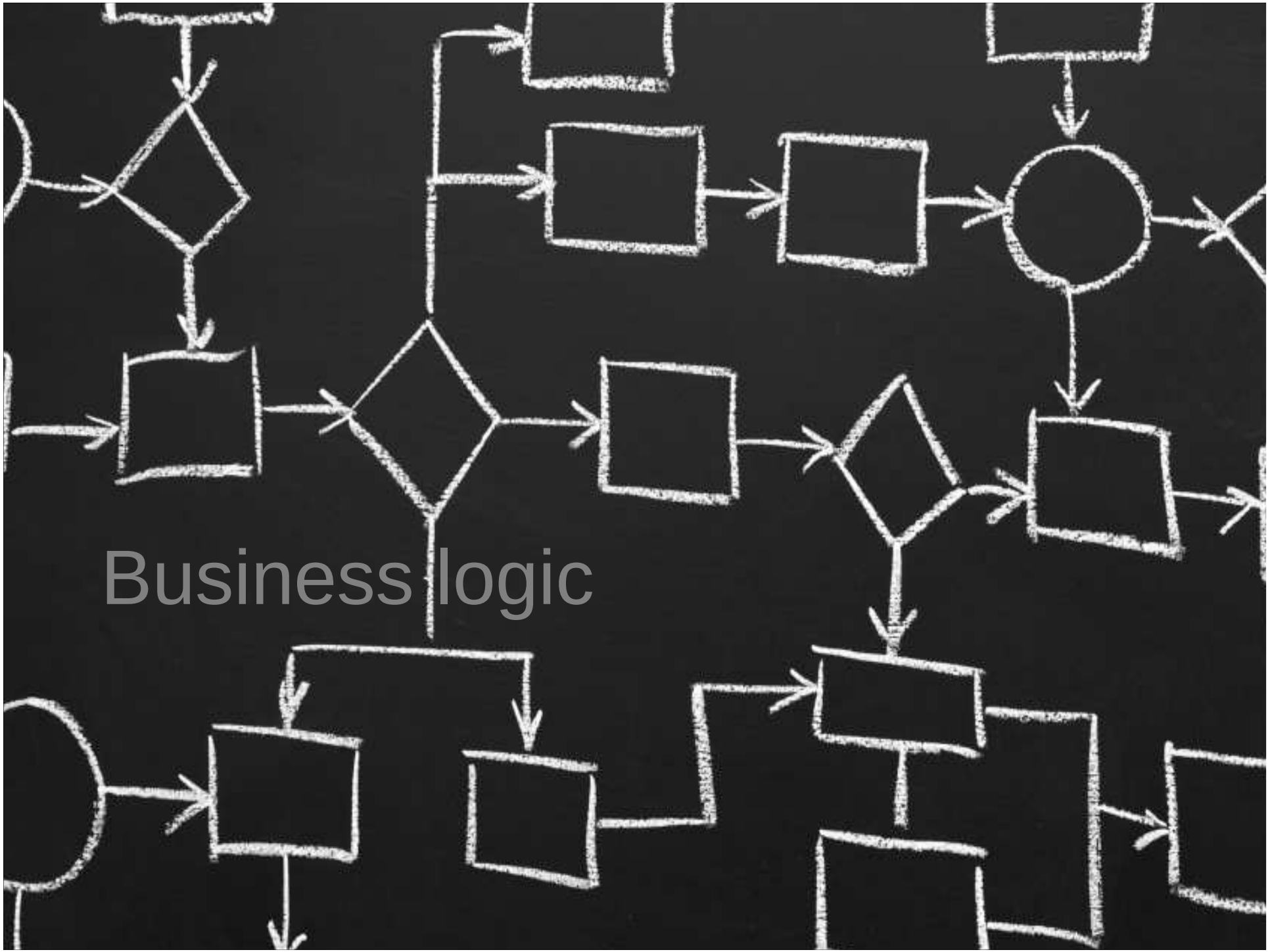
Frank Kleine, 27.10.2009

The Speaker: Frank Kleine

- 1&1 Internet AG
  (Head of Web Infrastructure)
- PHP since 2000
- Lead Developer
  - Stubbles
  - XJConf for PHP
  - vfsStream
- Technical editor "PHP Design Patterns"

Separation of Concerns

- Typical concerns in applications


Business logic


Logging

Error handling

Environment handling

Didn't we forget something really important?

Construction of Objects!

Construction of objects

- new is a very powerful keyword

Construction of objects

- new is a very powerful keyword
- Binds usage to a concrete implementation

Construction of objects

- new is a very powerful keyword
- Binds usage to a concrete implementation
- Golden rule of new:
  - OK for domain classes, but not for services

Construction of objects

- new is a very powerful keyword
- Binds usage to a concrete implementation
- Golden rule of new:
  - OK for domain classes, but not for services
  - OK in tests and specialised construction classes, but not in business logic

Construction of objects II

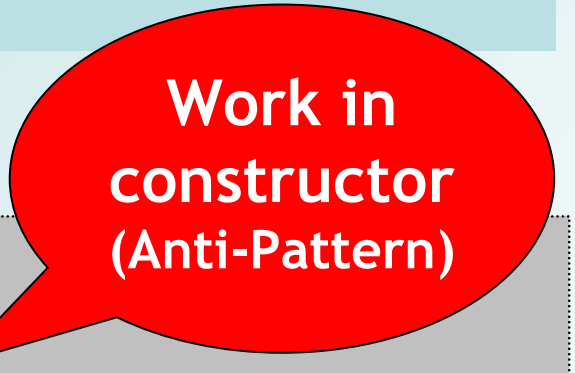
- Bad:

```
class Car {
    public function __construct() {
        $this->engine = new Engine();
        $this->tire   = TireFactory::createTire();
    }
}
```

Construction of objects II

- Bad:

```
class Car {
    public function __construct() {
        $this->engine = new Engine();
        $this->tire   = TireFactory::createTire();
    }
}
```



Construction of objects II

- Bad:

Hard to test

Work in constructor (Anti-Pattern)

```
class Car {
    public function __construct() {
        $this->engine = new Engine();
        $this->tire   = TireFactory::createTire();
    }
}
```

Construction of objects II

- Bad:

Hard to test

Work in constructor (Anti-Pattern)

```
class Car {
    public function __construct() {
        $this->engine = new Engine();
        $this->tire   = TireFactory::createTire();
    }
}
```

- Good:

```
class Car {
    public function __construct(Engine $eng, Tire $tire) {
        $this->engine = $eng;
        $this->tire   = $tire;
    }
}
```

Construction of objects II

- Bad:

Hard to test

Work in constructor (Anti-Pattern)

```
class Car {
    public function __construct() {
        $this->engine = new Engine();
        $this->tire   = TireFactory::createTire();
    }
}
```

- Good:

Piece of cake to test

```
class Car {
    public function __construct(Engine $eng, Tire $tire) {
        $this->engine = $eng;
        $this->tire   = $tire;
    }
}
```

Construction of objects II

- Bad:

Hard to test

Work in constructor (Anti-Pattern)

```
class Car {
    public function __construct() {
        $this->engine = new Engine();
        $this->tire   = TireFactory::createTire();
    }
}
```

- Good:

Piece of cake to test

```
class Car {
    public function __construct(Engine $eng, Tire $tire) {
        $this->engine = $eng;
        $this->tire   = $tire;
    }
}
```

Dependency Injection

Dependency Injection

- Pile of new (startup phase)
  - Create objects and stack them together
  - "Boilerplate", but automatable with DI framework
- Pile of objects (runtime phase)
  - Business logic, error handling, etc.
- Factories or DI instances for runtime object creation


CAUTION
Dependency Injection is viral

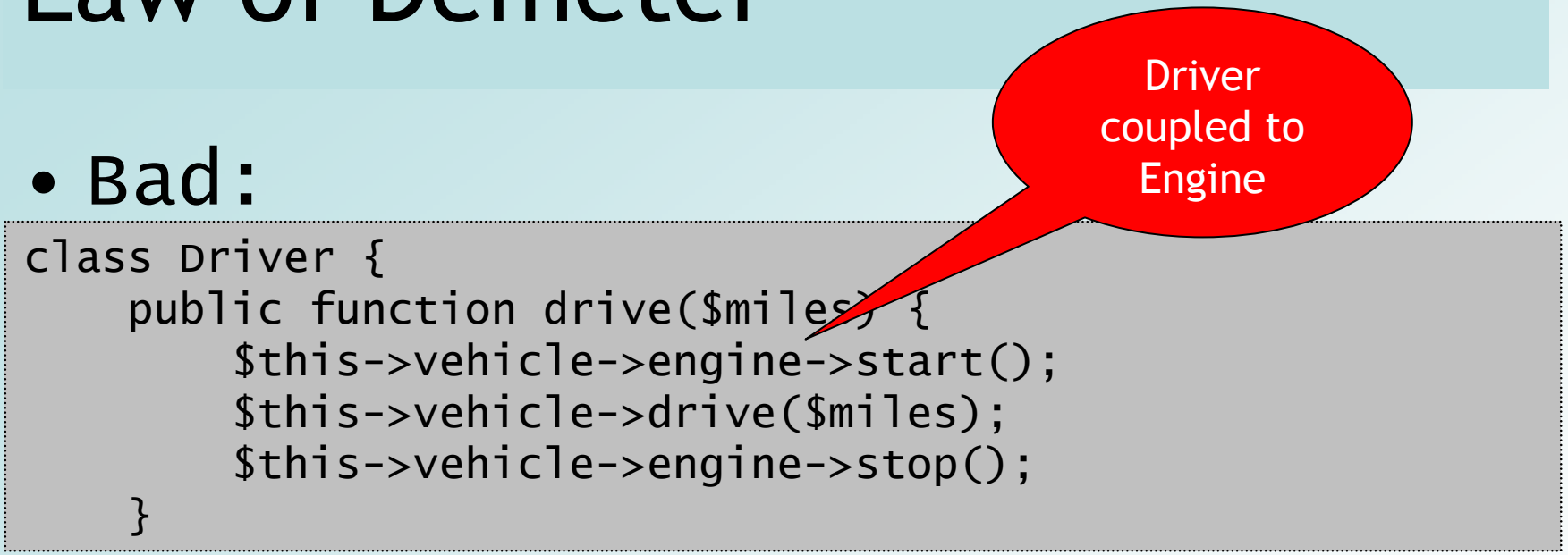
Law of Demeter

- Bad:

```
class Driver {
    public function drive($miles) {
        $this->vehicle->engine->start();
        $this->vehicle->drive($miles);
        $this->vehicle->engine->stop();
    }
```

Law of Demeter

- Bad:




```
class Driver {
    public function drive($miles) {
        $this->vehicle->engine->start();
        $this->vehicle->drive($miles);
        $this->vehicle->engine->stop();
    }
```

Law of Demeter

- Bad:

Hard to test: test always needs Engine or a mock of it

Driver coupled to Engine

```
class Driver {
    public function drive($miles) {
        $this->vehicle->engine->start();
        $this->vehicle->drive($miles);
        $this->vehicle->engine->stop();
    }
```

Law of Demeter

- Bad:

```
class Driver {
    public function drive($miles) {
        $this->vehicle->engine->
        $this->vehicle->drive(
        $this->vehicle->engine->
    }
```

Hard to test: test always needs Engine or a mock of it

Driver coupled to Engine

Internal state of Vehicle revealed

Law of Demeter

- Bad:

```
class Driver {
    public function drive($miles) {
        $this->vehicle->engine->
        $this->vehicle->drive(
        $this->vehicle->engine->
    }
```

Hard to test: test always needs Engine or a mock of it

Driver coupled to Engine

Internal state of Vehicle revealed

- Good:

```
class Driver {
    public function drive($miles) {
        $this->vehicle->start();
        $this->vehicle->drive($miles);
        $this->vehicle->stop();
    }
```

Law of Demeter

- Bad:

```
class Driver {
    public function drive($miles) {
        $this->vehicle->engine->
        $this->vehicle->drive(
        $this->vehicle->engine->
    }
```

Hard to test: test always needs Engine or a mock of it

Driver coupled to Engine

Internal state of Vehicle revealed

- Good:

```
class Driver {
    public function drive($miles) {
        $this->vehicle->start();
        $this->vehicle->drive($miles);
        $this->vehicle->stop();
    }
```

Driver not coupled to Engine: simpler to maintain

Law of Demeter

- Bad:

```
class Driver {
    public function drive($miles) {
        $this->vehicle->engine->
        $this->vehicle->drive(
        $this->vehicle->engine->
    }
```

Hard to test: test always needs Engine or a mock of it

Driver coupled to Engine

Internal state of Vehicle revealed

- Good:

```
class Driver {
    public function drive($miles) {
        $this->vehicle->start();
        $this->vehicle->drive($miles);
        $this->vehicle->stop();
    }
```

Piece of cake to test

Driver not coupled to Engine: simpler to maintain

Law of Demeter

- Bad:

```
class Driver {
    public function drive($miles) {
        $this->vehicle->engine->
        $this->vehicle->drive(
        $this->vehicle->engine->
    }
```

Hard to test: test always needs Engine or a mock of it

Driver coupled to Engine

Internal state of Vehicle revealed

- Good:

```
class Driver {
    public function drive($miles) {
        $this->vehicle->start();
        $this->vehicle->drive($miles);
        $this->vehicle->stop();
```

Piece of cake to test

Driver not coupled to Engine: simpler to maintain

Less prone to errors

Global state

- Same operation with same inputs should yield same results.
- Not necessarily true with global state.

Global state

- Same operation with same inputs should yield same results.
- Not necessarily true with global state.
- Hidden global state
  - Singleton
  - `static` methods
  - $_GET, $_POST, $_SESSION, $_...
  - Registry

Global state: Singletons

- Never implement any
- Required most likely due to bad design of your framework

Global state: Singletons

- Never implement any
- Required most likely due to bad design of your framework

- Use a DI framework with support for Singleton scope
  - Gives full power of Singletons
  - Code using the Singleton stays simple

Singleton with DI framework

```
$binder->bind('Session')
       ->to('PhpSession')
```

Singleton wit[Configure the binding] ...nework

```
$binder->bind('Session')
       ->to('PhpSession')
```

Singleton wit

```
$binder->bind('Session')
       ->to('PhpSession')
       ->in(stubBindingScopes::$SINGLETON);
```

Configure the binding

Enforce singletoness: DI framework will only create one instance of PhpSession and inject always this same instance

Singleton wit… n…

Configure the binding

Enforce singletoness: DI framework will only create one instance of PhpSession and inject always this same instance

```
$binder->bind('Session')
       ->to('PhpSession')
       ->in(stubBindingScopes::$SINGLETON);
```

```
class Processor {
    protected $session;
    /**
     * @Inject
     */
    public function __construct(Session $session) {
        $this->session = $session;
    }
    …
}
```

Singleton wit… n…

```
$binder->bind('Session')
       ->to('PhpSession')
       ->in(stubBindingScopes::$SINGLETON);
```

Configure the binding

Enforce singletoness: DI framework will only create one instance of PhpSession and inject always this same instance

```
class Processor {
    protected $session;
    /**
     * @Inject
     */
    public function __construct(Session $session) {
        $this->session = $session;
    }
    …
}
```

Dependency Injection

Singleton wit… n…

```
$binder->bind('Session')
       ->to('PhpSession')
       ->in(stubBindingScopes::$SINGLETON);
```

Configure the binding

Enforce singletoness: DI framework will only create one instance of PhpSession and inject always this same instance

```
class Processor {
    protected $se…
    /**
     * @Inject
     */
    public function __construct(Session $session) {
        $this->session = $session;
    }
    …
}
```

Tell DI framework to inject required parameters on creation of Processor

Dependency Injection

Singleton wit… n…

```
$binder->bind('Session')
       ->to('PhpSession')
       ->in(stubBindingScopes::$SINGLETON);
```

Configure the binding

Enforce singletoness: DI framework will only create one instance of PhpSession and inject always this same instance

```
class Processor {
    protected $se…
    /**
     * @Inject
     */
    public function __construct(Session $session) {
        $this->session = $session;
    }
    …
}
```

Tell DI framework to inject required parameters on creation of Processor

Dependency Injection

Piece of cake to test: independent of PhpSession

Global state: `static` methods

- Not always bad
  - Simple operations without dependencies
- Always bad if state is involved
  - Might return different results with same input.

Global state: $_GET, $_POST, …

- Ugly to test

Global state: $_GET, $_POST, …

- Ugly to test
- Rule: never access those vars directly in business logic

Global state: $_GET, $_POST, …

- Ugly to test
- Rule: never access those vars directly in business logic
- Abstract access to globals with classes
  - Remember: Singleton is evil

Global state: $_GET, $_POST, …

- Ugly to test
- Rule: never access those vars directly in business logic
- Abstract access to globals with classes
  - Remember: Singleton is evil
- Even better: interfaces

Global state: $_GET, $_POST, …

- Ugly to test
- Rule: never access those vars directly in business logic
- Abstract access to globals with classes
  - Remember: Singleton is evil
- Even better: interfaces
- Security: use a Request class which enforces filtering/validating input

Global state: Registry

- Using a DI framework: no registry required

Global state: Registry

- Using a DI framework: no registry required
- Without DI framework: use Registry for config values only

Global state: Registry

- Using a DI framework: no registry required
- Without DI framework: use Registry for config values only
- Did I already mentioned that Singletons are evil?

Global state: Registry

- Using a DI framework: no registry required
- Without DI framework: use Registry for config values only
- Did I already mentioned that Singletons are evil?
  - `static` methods are enough


Change is coming.


Modify
Change is coming.


Simplify
Modify
Change is coming.


Simplify
Improve
Modify
Change is coming.

Finally…

- If you remember only one little thing of this presentation it should be…


Singletons are really, really
(and I mean really)
DANGER
EXTREMELY FLAMMABLE
CAUTION

Singletons are really, really (and I mean really)

EVIL

The End

- Questions and comments?

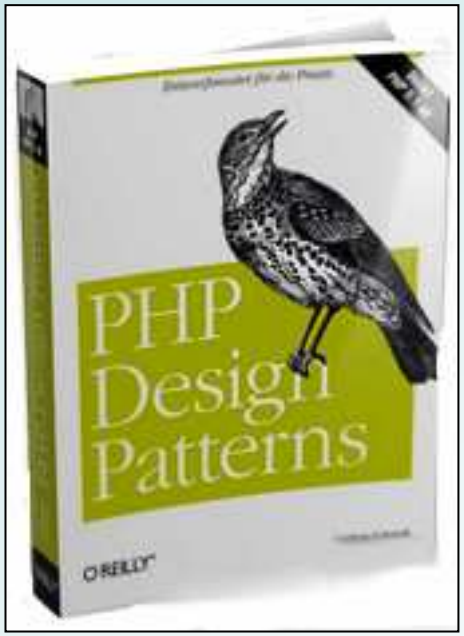
The End

- Questions and comments?

- Thank you.

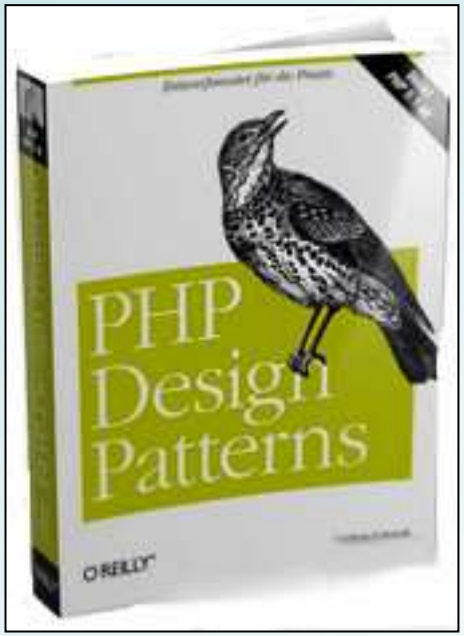
Commercial break

- Stephan Schmidt: PHP Design Patterns



Commercial break

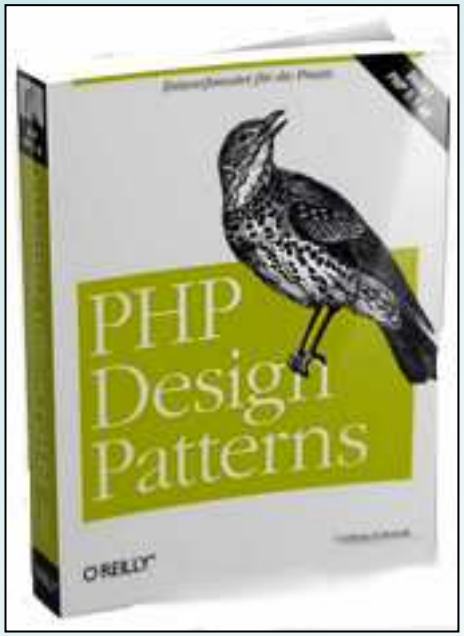
- Stephan Schmidt: PHP Design Patterns



- Clean code talks: http://www.youtube.com/view_play_list?p=79645C72EA595A91

Commercial break

- Stephan Schmidt: PHP Design Patterns



- Clean code talks: http://www.youtube.com/view_play_list?p=79645C72EA595A91
- Stubbles: www.stubbles.net