



Don't call us, we call you

Frank Kleine & Stephan Schmidt
1&1 Internet AG

ARRIVALS TO INTERNATIONAL PHP CONFERENCE				
DATE	ARRIVING FROM	FLIGHT	GATE	DESTINATION
03 11	NEW YORK	IPC07	A12	FRANKFURT
03 11	BERLIN	IPC07	A34	FRANKFURT
03 11	BUKAREST	IPC07	B45	FRANKFURT
03 11	TORONTO	IPC07	C90	FRANKFURT
03 11	PARIS	IPC07	C23	FRANKFURT
03 11	ROMA	IPC07	A78	FRANKFURT
04 11	LONDON			

Agenda

- Objektkomposition
- Verwenden von Factories
- Dependency Injection und Inversion of Control
- XJConf For PHP
- Stubbles IoC

Frank Kleine

- Senior Web-Developer bei der 1&1
- PHP-Entwickler seit 2000
- Lead Developer Stubbles
- Lead Developer XJConf for PHP
- Co-Autor "Exploring PHP"

Stephan Schmidt

- Leiter Web-Development bei der 1&1
- PHP-Entwickler seit 1999
- Entwickler in PEAR, PECL, pat, Stubbles
- Autor für verschiedene Fachmagazine
- Speaker auf internationalen Konferenzen
- Autor von PHP Design Patterns, erschienen im O'Reilly Verlag
- Leider heute krank

Objektkomposition

- Moderne Anwendungen bestehen aus sehr vielen Objekten
- Das Modell eines Autos könnte z.B. so aussehen

```
$bmw = new BMW($engine, $tire);
```

```
$bmw->moveForward(50);
```

Objektkomposition

- Besteht bereits aus vier Klassen

```
class Schst {...}
class Goodyear {...}
class TwoLitresEngine {...}

class BMW {
    ...
    public function __construct() {
        $this->engine = new TwoLitresEngine();
        $this->tire = new Goodyear();
        $this->driver = new Schst();
    }
    public function moveForward($miles) {
        $this->driver->sayHello();
        $this->engine->start();
        $this->tire->rotate();
    }
}
```

Probleme in diesem Entwurf

- Implementierung gegen konkrete Klassen
- Direkte Abhängigkeiten zwischen den Klassen (Klasse **BMW** erzeugt Instanzen im Konstruktor mit Hilfe des **new** Operators)
- Änderungen erfordern meistens Änderungen am bestehenden Sourcecode
- Klassen nicht austauschbar
- So gut wie nicht testbar

Einführen von Interfaces

```
interface Car {
    public function moveForward($miles);
}
interface Person {
    public function sayHello();
}
interface Tire {
    public function rotate();
}
interface Engine {
    public function start();
}
class Goodyear implements Tire {...}
class Schst implements Person {...}
class TwoLitresEngine implements Engine {...}
class BMW implementes Car {...}
```

Schon besser?

- Es gibt jetzt abstrakte Typen (durch Interfaces)
- Dennoch: Klasse **BMW** ist immer noch von den konkreten Klassen abhängig, da diese im Konstruktor verwendet werden.

*Keine deutliche Verbesserung
des Entwurfs*

Factories

- Verwendung von **new** durch Factory ersetzen

```
class BMW {  
    ...  
    public function __construct() {  
        $this->engine = CarFactory::getEngine();  
        $this->tire    = CarFactory::getTire();  
        $this->driver = CarFactory::getDriver();  
    }  
    ...  
}
```

- Factory erzeugt dann Instanzen der konkreten Klasse

Schon besser?

- Klasse **BMW** ist jetzt nicht mehr von **Goodyear**, **TwoLitresEngine** und **Schst** abhängig
- Factory-Methoden liefern Instanzen der abstrakten Typen **Tire**, **Engine** und **Person** zurück
- **Aber:** Klasse **BMW** ist jetzt von (**statischer**) Klasse **CarFactory** abhängig!
 - » Den Teufel mit dem Belzebub ausgetrieben

Das Hollywood Prinzip

"Rufen Sie uns nicht an, wir rufen Sie an."

- Auch als "*Inversion of Control*" oder "*Prinzip der Umkehrung der Abhängigkeiten*" bekannt
- Objekte sollen sich ihre Abhängigkeiten nicht selbst erzeugen oder holen
- Objekte sollen unabhängig von ihrer Umgebung (z.B. Factories) sein

Dependency Injection

- Abhängige Objekte werden von außen in die Objekte injiziert
- Objekte sind nicht mehr abhängig von konkreten Klassen oder der Umgebung, sondern nur von abstrakten Typen
- Abhängige Objekte sind sehr leicht austauschbar
 - neue Funktionalität (z.B. Strategy-Pattern)
 - Einfacher zu Testen (Mocks injizieren)

Dependency Injection

Constructor-Injection

```
class BMW implements Car {  
    ...  
    public function __construct(Engine $eng, Tire $tire) {  
        $this->engine = $engine;  
        $this->tire = $tire;  
    }  
    ...  
}  
  
// Client-Code  
$tire    = new Goodyear();  
$engine  = new TwoLitresEngine();  
  
$bmw     = new BMW($engine, $tire);
```

Dependency Injection

Setter-Injection

```
class BMW implements Car {  
    ...  
    public function setDriver(Person $driver) {  
        $this->driver = $driver;  
    }  
    ...  
}  
  
// Client-Code  
$driver = new Schst();  
$bmw    = new BMW(...);  
$bmw->setDriver($driver);
```

Ergebnis

- Die Klasse **BMW** ist nur von den Interfaces **Person**, **Tire** und **Engine** abhängig
- Keine Abhängigkeit von konkreten Klassen oder der Umgebung

Aber: Sehr viel zusätzlicher Client Code, durch Instanziieren und Injizieren der Objekte nötig.

DI Frameworks

- Häufig wiederkehrende Aufgabe verlangt nach Framework
- In der Java-Welt bereits stark etabliert
 - Spring, Picco Container, Google Guice, ...
- In der PHP Welt noch sehr wenig Frameworks
 - Stubbles, Garden, XJConf For PHP, Saesar
- Häufig Insellösungen pro Framework

XJConf For PHP

- XML-to-Object-Mapper
- Portierung der Java-Version von XJConf, welche wiederum durch patConfiguration inspiriert wurde
- Objekte werden mit XML-Tags repräsentiert
- XML-Schachtelungen ergeben Objektkompositionen

XJConf: Definitionen

- XJConf muss zunächst lernen, welche XML-Tags es gibt und was sie bedeuten:

```
<defines>
  <abstractTag name="car" abstractType="Car"
concreteTypeAttribute="type">
  <constructor>
    <child name="engine"/>
    <child name="tire"/>
  </constructor>
</abstractTag>
  <abstractTag name="person" abstractType="Person"
concreteTypeAttribute="type" setter="setDriver"/>
  <abstractTag name="tire" abstractType="Tire"
concreteTypeAttribute="type"/>
  <abstractTag name="engine" abstractType="Engine"
concreteTypeAttribute="type"/>
</defines>
```

XJConf: Objektkonfiguration

- Jetzt kann XJConf auch die Konfigurationsdatei lesen:

```
<configuration>  
  <car type="BMW">  
    <engine type="TwoLitresEngine"/>  
    <tire type="Goodyear"/>  
    <person type="Schst"/>  
  </car>  
</configuration>
```

XJConf: Objekt erzeugen

- Plain PHP:

```
$xjconf = new XJConfFacade();  
// Definitionen hinzufügen  
$xjconf->addDefinition('xjconf-defines.xml');  
// Konfiguration parsen  
$xjconf->parse('xjconf-config.xml');  
// Instanz holen  
$bmw = $xjconf->getConfigValue('car');  
$bmw->moveForward(50);
```

XJConf For PHP

- Interfaces, abstrakte Klassen: konkrete Typen erst zur Konfigurationszeit angeben
- „normale“ Klassen
- simple Typen, Arrays
- Fabrikmethoden
- statische Methoden
- Klassen zur Laufzeit nach Bedarf laden
- Erweiterbar durch Extensions
- Einführungstutorial auf IBM DeveloperWorks (verlinkt auf XJConf-Website)

XJConf For PHP

✓ Vorteile

- gut geeignet für Konfigurationsdateien, welche von nicht-Entwicklern gepflegt werden müssen (z.B. Variantenmanagement)
- Boilerplate-Code durch XML-Datei ersetzt

✗ Nachteile

- Boilerplate-Code durch XML ersetzt
- Definitionen für Objektmapping müssen gepflegt werden

Stubbles IoC Container

- Portierung von Google Guice (Java)
<http://code.google.com/p/google-guice/>
- Entwickelt als Teil des Stubbles Frameworks
- Kann auch außerhalb des Frameworks verwendet werden
- Download von
<http://www.stubbles.net>

Stubbles IoC Container

- Komplette anderer Ansatz als XJConf
 - Keine zusätzliche Konfigurationsdatei
 - (Fast) keine benannten Instanzen
- Basiert auf Type Hints und Annotations
 - Keine native Unterstützung in PHP
 - Annotations als Teil des Docblocks

```
/**  
 * Doc-Comment...  
 *  
 * @Inject  
 */
```

Interfaces binden

- Typen werden durch **Binder** an konkrete Implementierungen gebunden:

```
$binder = new stubBinder();  
$binder->bind('Tire')->to('Goodyear');
```

- Binder kann **Injector** liefern, der Instanzen erzeugt:

```
$injector = $binder->getInjector();  
$tire     = $injector->getInstance('Tire');  
  
// Implizites Binding  
$goodyear = $injector->getInstance('Goodyear');
```

Wo bleibt die Injection?

- Unterstützt Setter- und Constructor- Injection
- **@Inject** Annotation muss an die Methoden geschrieben werden, in die weitere Objekte injiziert werden sollen
- Type-Hints sind zwingend erforderlich
- Stubbles erkennt an Hand der Type-Hints den benötigten Typ und sucht das passende Binding

@Inject

```
class BMW implements Car {  
    /**  
     * Create a new BMW  
     *  
     * @Inject  
     */  
    public function __construct(Engine $e, Tire $t) {  
        $this->engine = $e;  
        $this->tire    = $t;  
    }  
    /**  
     * Set the driver  
     *  
     * @Inject  
     */  
    public function setDriver(Person $driver) {  
        $this->driver = $driver;  
    }  
}
```

@Inject

```
$binder = new stubBinder();

// Alle Interfaces an konkrete Typen binden
$binder->bind('Car')->to('BMW');
$binder->bind('Tire')->to('Goodyear');
$binder->bind('Person')->to('Schst');
$binder->bind('Engine')->to('TwoLitresEngine');

// Injector erzeugen
$injector = $binder->getInjector();

// Objekt über Interface-Namen holen
$bmw = $injector->getInstance('Car');

$bmw->moveForward(50);
```

Optionale Injection

- Ist an ein Interface keine Implementierung gebunden, wirft Stubbles eine Exception
- Injection kann auch als optional markiert werden

```
/**  
 * Set the driver  
 *  
 * @Inject(optional=true)  
 */  
public function setDriver(Person $driver) {  
    $this->driver = $driver;  
}
```

Standardimplementierung

- Häufig wird IoC nur gebraucht, um in Tests Mock-Objekte zu verwenden
- Ständiges Binding an die selbe Klasse kann vereinfacht werden

```
/**
 * Person interface
 *
 * @ImplementedBy(Schst.class)
 */
interface Person {
    public function sayHello();
}
```

Reihenfolge der Bindings

1. Überprüfen, ob Binding programmatisch hinzugefügt wurde (**`$binder->bind()`**)
2. Überprüfen, ob **`@ImplementedBy`** Annotation vorhanden ist
3. Verwenden des impliziten Bindings, falls der Typ eine instanziiierbare Klasse ist

Binden an Instanz

- Typ an an bestehende Instanz einer Klasse gebunden werden:

```
$tire = new Goodyear();  
$binder->bind('Tire')->toInstance($tire);
```

Unterstützung für Singletons

- Jede Klasse kann als Singleton verwendet werden:

```
$binder->bind('Tire')->to('Goodyear')  
    ->in(stubBindingScopes::$SINGLETON);
```

- Auch als Annotation möglich

```
/**  
 * @Singleton  
 */  
class Schst implements Person {  
    ...  
}
```

Benennung von Instanzen

- Mehrere Instanzen des selben Typs können über Benennung unterschiedlich behandelt werden

```
class BMW implements Car {  
    /**  
     * @Inject  
     * @Named('Driver')  
     */  
    public function setDriver(Person $driver) {  
        $this->driver = $driver;  
    }  
}
```

Benennung von Instanzen

- Name wird beim Binding angegeben

```
$binder->bind('Person')->named('Driver')->to('Schst');  
$binder->bind('Person')->named('Co-Driver')->to('Mikey');  
  
// Alle Person Instanzen ohne Namen  
$binder->bind('Person')->to('User');
```

- Kann mit anderen Features kombiniert werden

```
$binder->bind('Person')  
  ->named('Driver')  
  ->to('Schst')  
  ->in(stubBindingScopes::$SINGLETON);
```

Injizieren von skalaren Werten

Nur mit Setter-Injection

```
class Person (  
  protected $name;  
  /**  
   * @Inject  
   * @Named('personName')  
   */  
  public function setName($name) {...}  
)
```

```
$binder->bindConstant()->named('personName')  
  ->to('Stephan Schmidt');  
$p = $binder->getInjector()->getInstance('Person')
```

Erweiterbarkeit

- Eigene Provider zum Erstellen von Objekten, die nicht über den Injector erstellt werden können
 - » ideal zur Integration von Factories
- Eigene Scopes (analog zu Singleton), um die Gültigkeit der Instanzen zu beschränken
 - » z.B. Objekt pro User, das in einem Cache liegt

Stubbles IoC

✓ Vorteile

- Keine zusätzliche Konfiguration
- Sehr wenig zusätzliche Arbeit
- Sehr intuitiv zu bedienen
- Sehr einfach zu erweitern

✗ Nachteile

- Bindung an konkretes IoC Framework (Annotations stehen im Quellcode)
- Bei vielen unterschiedlichen konkreten Implementierungen unübersichtlich

Weitere Frameworks

- Garden
 - 1:1 Portierung des IoC-Teils von Spring
 - Konfiguration der Objekte mittels XML
 - Seit 18 Monaten keine Entwicklung mehr
- S2Container.PHP5
 - Portierung von Seasar2 aus Java
 - Konfiguration der Objekte mittels XML
 - Große Ähnlichkeit mit Spring
 - Unterstützt auch AOP

Fazit

- Dependency Injection macht Anwendungen erweiterbarer, testbarer und sorgt für saubere Architekturen
- Verschiedene Ansätze in PHP bereits vorhanden
- Auswahl des Frameworks an Anforderungen und persönliche Präferenzen anpassen

Danke!

Vielen Dank für Ihre Aufmerksamkeit.

Noch Fragen?

php@frankkleine.de

schst@stubbles.net

Links

- Slides: <http://stubbles.net/>
- XJConf: <http://php.xjconf.net/>
- Stubbles: <http://stubbles.net/>
- Google Guice (Java):
<http://code.google.com/p/google-guice/>
- Garden: <http://garden.tigris.org/>
- S2Container.PHP5:
[http://s2container.php5.seasar.org/index_en.](http://s2container.php5.seasar.org/index_en)