Getting started with Dependency Injection

Rob Allen
May 2015

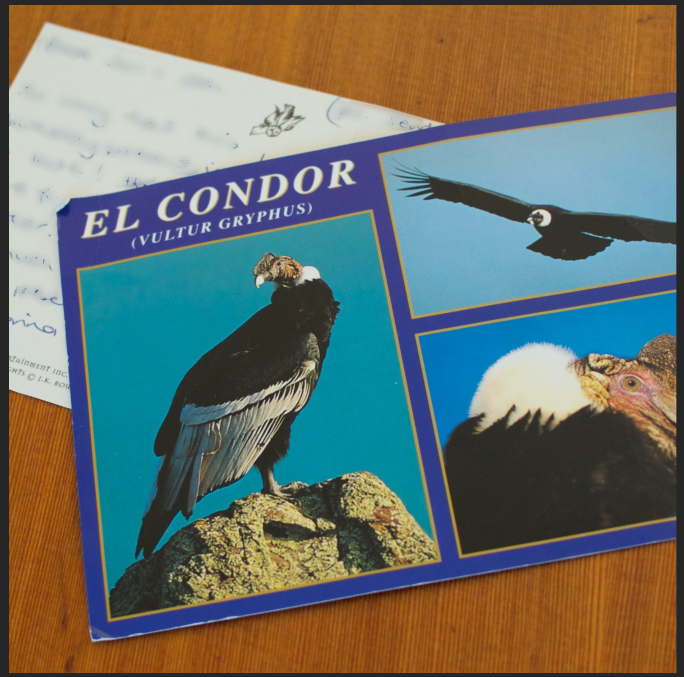
http://19ft.com

Dependency Injection enables loose coupling and loose coupling makes code more maintainable

Mark Seemann

We're actually talking about
loose coupling today

Coupling



Benefits of loose coupling

- Maintainability - Classes are more clearly defined
- Extensibility - easy to recompose application
- Testability - isolate what you’re testing

A worked example

Class A needs class B in order to work.

```
class Letter
{
    protected $paper;

    public function __construct()
    {
        $this->paper = new WritingPaper();
    }
}

// usage:
$letter = new Letter();
$letter->write("Dear John, ...");
```

Pros and cons:

Pros:

- Very simple to use

Cons:

- *Tight coupling*
    - Cannot test `Letter` in isolation
    - Cannot change `$paper`

How do we change the paper size?
How do we change the type of paper?

Method parameters?

```
class Letter
{
    protected $paper;

    public function __construct($size)
    {
        $this->paper = new WritingPaper($size);
    }
}

// usage:
$letter = new Letter('A4');
$letter->write("Dear John, ...");
```

Use a Registry?

```
class Letter
{
    protected $paper;

    public function write($text)
    {
        $paper = Zend_Registry::get('paper');
        return $paper->placeWords($text);
    }
}
```

Use a Registry?

```
// usage:
Zend_Registry::set('paper', new AirmailPaper('A4'));

$letter = new Letter();
$letter->write("Dear John, ...");
```

Inject the dependency!

Injection

```
class Letter
{
    protected $paper;

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

Injection

```
// usage:
$letter = new Letter(new WritingPaper('A4'));
$letter->write("Dear John, ...");
```

This is also known as

Inversion of Control

Pros and cons:

Pros:

- Decoupled `$paper` from `Letter`:
    - Can change the type of paper
    - Natural configuration of the `Paper` object
- Can test `Letter` independently

Cons:

- Burden of construction of `$paper` is on the user

Dependency Injection

That's it; we're done!

Types of injection

Constructor injection:

```
$letter = new Letter($paper);
```

Setter injection:

```
$letter = new Letter();
$letter->setPaper($paper);
```

Property injection:

```
$letter = new Letter();
$letter->paper = $paper;
```

Interface injection:

```
interface InjectPaper
{
    public function injectPaper($paper);
}

$letter = $container->get('Letter');
// container automatically calls
// injectPaper() if Letter implements
// InjectPaper
```

Note

Too many constructor parameters is a *code smell*

Two-phase construction is *Bad(TM)*

Rules of thumb

- Limit the scope of any given object
- Depend on interfaces
- Use constructor injection

How about usage?

```
$paper = new AirmailPaper('A4');
$envelope = new Envelope('DL');
$letter = new Letter($paper, $envelope);

$letter->write("Dear John, ...");
```

Setup of dependencies gets tedious quickly

Dependency Injection Container

An object that handles the creation of objects and their dependencies

Dependency resolution can be *automatic* or *configured*

DICs are *optional*

Write a simple container

```
class LetterContainer
{
    public function getLetter()
    {
        $paper = new AirmailPaper('A4');
        $envelope = new Envelope('DL');
        $letter = new Letter($paper, $envelope);

        return $letter;
    }
}
```

Usage

```
$container = new LetterContainer()
$letter = $container->getLetter();
```

Handle configuration

```
class LetterContainer
{
    protected $params;

    public function __construct(array $params)
    {
        $this->params = $params;
    }
```

cont…

```
    public function getLetter()
    {
        $paper = new AirmailPaper(
            $this->params['paper.size']);
        $envelope = new Envelope(
            $this->params['envelope.size']);

        $letter = new Letter($paper, $envelope);

        return $letter;
    }
}
```

Usage

```
// usage:
$container = new LetterContainer(array(
    'paper.size' => 'A4',
    'envelope.size' => 'DL',
    ))
$letter = $container->getLetter();
```

Now, it’s easy to change parameters of the dependent objects

Shared objects

```
class LetterContainer
{
    protected $shared;

    public function getLetter()
    {
        if (!isset(self::$shared['letter'])) {
            // ... create $letter as before ...
            self::$shared['letter'] = $letter;
        }
        return self::$shared['letter'];
    }

    // ...
```

Dependency Injection Container

- Creates objects on demand
- Manages construction of an object’s dependencies
- Separates of configuration from construction
- Can allow for shared objects

However:

Writing and maintaining a container class by hand is *tedious*!

Available DICs

Don’t reinvent the wheel

- *Aura.Di* - part of Aura
- *Container* - part of “The League”
- *Dice* by Tom Butler
- *Illuminate Container* - part of Laravel
- *PHP-DI* by Matthieu Napoli
- *Pimple* by Fabien Potencier
- *Service Container* - part of Symfony2
- *Zend\Di & Zend\ServiceManager* - part of ZF 2

Pimple

- Easy to use
- Small: only 179 lines of PHP
- Configured manually

Pimple

```
$container = new Pimple();

$container['letter'] = function ($c) {
    $paper = new AirmailPaper('A4');
    $envelope = new Envelope('DL');

    $letter = new Letter($paper, $envelope);
    return $letter;
};
```

Pimple usage

```
$container = new Pimple();
$letter = $container['letter'];
```

More typically

```
$container = new Pimple();

$container['paper.size'] = 'A4';
$container['envelope.size'] = 'DL';

$container['paper'] = function ($c) {
    $size = $c['paper.size'];
    return new AirmailPaper($size);
};

$container['envelope'] = function ($c) {
    $size = $c['envelope.size'];
    return new Envelope($size);
};
```

cont…

```
$container['letter'] = function ($c) {
    $paper = $c['paper'];
    $envelope = $c['envelope'];
    return new Letter($paper, $envelope);
};
```

Usage is identical:

```
$container = new Pimple();
$letter = $container['letter'];
```

Too much PHP?!

Symfony’s Container supports YAML!

services.yml

```
parameters:
    paper.size: A4

services:
    paper:
        class:     AirmailPaper
        arguments: ["%paper.size%"]
    letter:
        class:     Letter
        arguments: ["@paper"]
```

Usage

```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;

// configure container
$container = new ContainerBuilder();
$locator = new FileLocator(__DIR__);
$loader = new YamlFileLoader($container, $locator);
$loader->load('services.yml');

// Retrieve from container
$letter = $container->get(letter');
```

Recommendation

- Hold configuration separately
- Create each each object separately

Automatic resolution

```
class Letter
{
    protected $paper;

    public function __construct(AirmailPaper $paper)
    {
        $this->paper = $paper;
    }
}
```

Automatic resolution

```
$di = new Zend\Di\Di();
$letter = $di->get('Letter');
var_dump($letter);
```

Output:

```
class Letter#18 (2) {
  protected $paper =>
  class AirmailPaper#17 (1) {
    private $size =>
    string(2) "A4"
  }
}
```

Zend\Di configuration

via instanceManager:

```
$im = $di->instanceManager();
$im->setParameters('AirmailPaper', ['size' => 'A4']);
$letter = $di->get('Letter');
```

when retrieving:

```
$letter = $di->get('Letter', array('size' => 'A4'));
```

Interface dependencies

```
class AirmailPaper implements PaperInterface { /* … */}
class Letter {
  function __construct(PaperInterface $paper)) { /* … */}
}

$di = new Zend\Di\Di();
$im = $di->instanceManager();
$im->setParameters('AirmailPaper', ['size' => 'A4']);
$im->addTypePreference('PaperInterface', 'AirmailPaper');

$letter = $di->get('Letter');
```

Note

- Automatic resolution can be much slower
- “Magic” is happening!

There’s a reason that most DICs are configured.

A note on Service Location

Service Location

```
class Letter
{
    protected $paper;

    public function __construct($locator)
    {
        $this->paper = $locator->get('paper');
    }
}
```

Service Location

- Component *pulls* its dependencies in when it needs them
- Still decouples concrete implementation of `Paper` from `Letter`

however

- Hidden dependencies! Who knows what is being used?
- Makes testing harder

Recap

Dependency injection promotes:

- Loose coupling
- Separation of configuration from construction from usage
- Easier testing

“Dependency Injection” is a 25-dollar term for a 5-cent concept.

James Shore

Thank you!

https://m.joind.in/talk/3d82c

Rob Allen - http://akrabat.com - @akrabat