



A Dependency Injection Primer

In which [@rowan_m](#) discovers a gateway drug to quality.



Chapter 1 ~ The Green Field

*In which our heroine begins a new
endeavour for a local publican*

Developers shouldn't couple

How one might start to write their application code:

```
$bar = new DrinkMapper();  
$drink = $bar->getCocktail('bloody mary');
```



Naïve use of patterns

Supporting code as written by our eager, young protagonist:

```
// our data mapper  
class DrinkMapper {  
    public function getCocktail($name) {  
        $bar = new PitcherAndPiano();  
        $row = $bar->getCocktail($name);  
        return new Cocktail($row);  
    }  
}  
  
// our domain object  
class Cocktail extends ArrayObject {}
```

Naïve use of patterns (continued)

Supporting code as written by our eager, young protagonist:

```
// our storage
class PitcherAndPiano {
    public function getCocktail($name) {
        $db = new PDO('mysql:host=localhost;'.
            'dbname=php', 'puser', 'ppass');
        $row = $db
            ->query("SELECT * FROM cocktails'.
                ' WHERE name = '$name'")
            ->fetch();
        return $row;
    }
}
```



Chapter 2 ~ Foreboding Clouds

*In which the gruff senior developer
mutters darkly about unit tests*

Constructors for a house of cards

Our heroine attempts to test from the deepest class upwards.

~

Emboldened by her success with **PHPUnit_Extensions_**
Database_TestCase on the **PitcherAndPiano** class, she moves on...



Foundations begin to crumble

This method ~ cannot ~ be unit tested due to these concrete dependencies within the code.

```
// our data mapper
class DrinkMapper {
    public function getCocktail($name) {
        $bar = new PitcherAndPiano();
        $row = $bar->getCocktail($name);
        return new Cocktail($row);
    }
}
```

Filling in the cracks

After some research, our intrepid heroine tries Setter Injection:

```
class DrinkMapper {  
    private $_bar;  
  
    public function setBar(  
        PitcherAndPiano $bar) {  
        $this->_bar = $bar;  
    }  
  
    public function getCocktail($name) {  
        $row = $this->_bar->getCocktail($name);  
        return new Cocktail($row);  
    }  
}
```

Setter Injection Abuse

*Still the saturnine senior shows scorn
for the implied improvements.*

~

Setters should be used with caution as they introduce mutability.

Immutable classes are simpler and more efficient.

~

This setter requires that it is called before the finder method.
The object cannot be constructed in a valid state.

Constructing correctly

*Enlightened, our heroine progresses to **Constructor Injection**:*

```
class DrinkMapper {  
    private $_bar;  
  
    public function setBar__construct(  
        PitcherAndPiano $bar) {  
        $this->_bar = $bar;  
    }  
  
    public function getCocktail($name) {  
        $row = $this->_bar->getCocktail($name);  
        return new Cocktail($row);  
    }  
}
```

Understand rules to break them

Of course, our senior now provides contrary observations.

~

A long list of constructor parameters may indicate the class has too many responsibilities. However, it may be valid. Setter injection is the right solution in places.

~

Constructor parameters are not self-documenting. One must refer to the code of the called class.



Chapter 3 ~ Weather the Storm

In which our heroine puts her new skills into practice via a training montage.

The other class

Our heroine ponders why her new skills do not fit the other class.

```
class DrinkMapper {
    private $_bar;

    public function __construct(
        PitcherAndPiano $bar) {
        $this->_bar = $bar;
    }

    public function getCocktail(
        $name, Cocktail $cocktail) {
        $row = $this->_bar->getCocktail($name);
        return $cocktail->exchangeArray($row);
    }
}
```

An awkward feeling

Without disturbing the senior's slumber, our heroine ponders...

~

That is now dependent on another mutable class. It does not make sense for a **Cocktail** to exist sans content.

~

Callers must now construct the return type too, leading to a great deal of repeated code.

The industrial revolution

Inspiration strikes and our heroine recalls the Factory Pattern:

```
class DrinkFactory {  
    public static $ctClass = 'Cocktail';  
  
    public static function getCocktail(array $data) {  
        return new self::$ctClass($data);  
    }  
}
```

```
class DrinkMapper { [...]  
    public function getCocktail($name) {  
        $row = $this->_bar->getCocktail($name);  
        return DrinkFactory::getCocktail($row);  
    }  
}
```

Mass production

With a few tweaks, our heroine obtains her holy grail of 100% coverage!

~

Factories can replace constructors within libraries. Controllers may arguably use constructors directly due to their specialised nature.

~

Factories may also detect the presence of an object over a class name to aid in injecting mock objects.



Chapter 4 ~ Pastures New

*In which our heroine travels on her
new wheel, only to discover...
she did not invent it first!*

Exploring the frontier

Our heroine begins to push the boundaries.

~

Define interfaces for injected classes for even looser coupling.

~

Swap implementations in real-life, not just mocks.

Founding fathers

Martin Fowler first coined the term as a specialisation of Inversion of Control:

<http://martinfowler.com/articles/injection.html>

~

He additionally covers Interface Injection and Service Locators

~

Goodness, he's a sharp fellow:

<http://twitter.com/martinfowler>



They are among us!

Richard Miller has written rather extensively on the subject with respect to Symfony 2:

<http://miller.limethinking.co.uk/tag/dependency-injection/>

~

Take advantage of his delicious brain.

~

He may also be found on Twitter:

http://twitter.com/mr_r_miller





Questions & Discussion

~ *fin* ~