php|tek - Chicago, May 16-18, 2007

Dependency Injection

Modularity for Reuse

Jeff Moore
http://www.procata.com/

Interdependence

* Technology has made us wealthy
* Nobody in our society is self-sufficient
* We specialize
* We build on the work of others

Source of Material Wealth

* Interchangeable parts
* Mass production
* Machine tools

A Tale of Two Clocks

- Master Clockmaker
  - Well organized
  - More reliable
- Apprentice
  - Haphazard
  - Lesser quality

A Modern Clock

* Cheap
* Reliable
* Available
* Predictable
* Ugly?

Software Craftsmanship

* Each piece largely unique
* Large variation in quality between master and apprentice

Software Industrialization

* Linux, Apache, MySQL, and PHP
* PHP is a thin layer over many C libraries
* PEAR or Smarty
* Drupal or WordPress

Reusable Code

* Interchangeable Parts
* Mass Production
* Machine Tools

Craftsmanship is here to stay

* Higher level code is harder to reuse
* Reuse requires customization and configuration

Drawbacks to Reuse

Your Fate in Someone Else's Hands

* Reusing code introduces risky dependencies
* Change in direction
* Abandonment
* Missing new technologies
* Infrequent updates

Reasons Not to Reuse

* Doesn't do exactly what you need
* It does too much
* Not extensible
* Not configurable
* Lack of support
* Lack of documentation

Good Code Is Hard to Find

* Evaluation costs are high
* Few packages with reputation for quality and reusability
* Hard to separate the wheat from the chaff

Producing Reusable Code is Hard

* Solves a problem
* Simple
* Configurable
* Documented
* Flexible

How to Promote Reuse?

Dependency Injection

What is a Dependency?

* You can't use that without this
* I come with strings attached

A Tangled Web Of Dependencies

FeedFetcher

* `Class FeedFetcher {}`
* Fetch a feed, given an Url
* This could be a performance bottle neck?

FeedCache

```
class FeedFetcher {
  protected $cache;
  function __construct() {
    $this->cache =
        new FeedCache();
  }
}
```

FeedCache Configuration

* Maximum age
* Cache directory
* Error handling

How do we Configure FeedCache?

```
class FeedFetcher {
  protected $cache;
  function __construct() {
    $this->cache =
        new FeedCache();
  }
}
```

Introduce Constants?

* `define('FEED_CACHE_DIR', '/tmp');`
* `define('FEED_CACHE_AGE', 3600);`
* `define('FEED_CACHE_DEBUG', true);`

The Problems with Constants

* What if you need different configuration options for different feeds?
* Global

How do we Configure FeedCache?

```
class FeedFetcher {
  protected $cache;
  function __construct() {
    $this->cache =
        new FeedCache();
  }
}
```

Add Cache Configuration Options to FeedFetcher

```
class FeedFetcher {
  protected $cache;
  function __construct($cacheDir, $maxAge) {
    $this->cache =
        new FeedCache($cacheDir, $maxAge);
  }
}
```

Bad API

* Bloated interface for FeedFetcher
* Can't use FeedFetcher without FeedCache

How Do We Have More Than One Kind of Cache?

```
class FeedFetcher {
  protected $cache;
  function __construct() {
    $this->cache =
      new FeedCache();
  }
}
```

Possible Cache Classes

* File System Cache
* Database Cache
* Shared Memory Cache
* Null Cache

The Driver Pattern

```
class FeedCache {
  static function getCache(
      $driver,
      $options = array()) {

    $file = '/drivers/' . $driver . '.php';
    require_once $file;
    $classname = ucfirst($driver) . 'Cache';
    return new $classname($options);

  }
}
```

Defining the Driver

```
class DriverCommon {
  protected $options;
  function __construct($options) {
    $this->options = $options;
  }
}

class FileCache extends DriverCommon {}
```

Using the Driver

```
class FeedFetcher {
  protected $cache;
  protected $options;

  function __construct($options = array()) {
    $this->cache = FeedCache::getCache(
      $options['driver'],
      $options);
  }

}
```

Still have a problem with Configuration Options

* FeedFetcher API is still coupled to the caching configuration options
* FeedCache receives configuration options it might not care about
* Amorphous options array complicates both APIs

Difficult to Test or Extend

* Mapping between driver name, file and class name
* Hard to inject an arbitrary driver class
* Each mock object would require a physical file

How to Inject a Mock FeedCache for Testing?

```
class FeedFetcher {
  protected $cache;
  function __construct() {
    $this->cache =
        new FeedCache();
  }
}
```

Factory Methods?

```
class FeedFetcher {
  protected $cache;

  function __construct() {
    $this->cache = $this->createCache();
  }

  function createCache() {
  }
}
```

Use Inheritance?

```
class MyFeedFetcher extends FeedFetcher {

  function createCache() {
    return new FileFeedCache('/tmp');
  }

}
```

Inheritance ... Not

* Heavy weight
* Wastes your one chance at single inheritance
* Cumbersome API

Untangling Dependencies

How to Configure FeedCache?

```
class FeedFetcher {
  protected $cache;
  function __construct() {
    $this->cache =
        new FeedCache();
  }
}
```

Bad Options

* Configure FeedCache through FeedFetcher
* Configure FeedCache around FeedFetcher
* Extend FeedFetcher to configure FeedCache

Turn the Relationship Inside Out

Don't pull the dependency In

```
class FeedFetcher {
  protected $cache;
  function __construct() {
    $this->cache =
        new FeedCache();
  }
}
```

Inject or Push the dependency In

```
class FeedFetcher {
  protected $cache;
  function __construct($cache) {
    $this->cache = $cache;
  }
}
```

Configuration on the Outside

```
$cache = new FileCache('/tmp');
$fetcher = new FeedFetcher($cache);
```

Factor Out an Interface

```
interface Cache {}

class FileCache implements Cache {}
class MemoryCache implements Cache {}
class DbCache implements Cache {}
```

Depend on the interface, not the Class

```
class FeedFetcher {
  protected $cache;
  function __construct(Cache $cache) {
    $this->cache = $cache;
  }
}
```

Use Adapters to Plugin Third Party Cache

```
class CacheAdapter
    extends ThirdPartyCache
    implements Cache {
}

new FeedFetcher(new CacheAdapter(...));
```

FeedFetcher is Now Easy To Test

* `new FeedFetcher(new NullCache());`
* No cache configuration required to test FeedFetcher
* No extension or modification to FeedFetcher required for testing

Explicit Dependencies

* Hidden, implicit dependencies make code hard to test and hard to reuse
* Explicitly declare dependencies on the Interface of the class
* Easier to maintain

Alternatives To Dependency Injection

Service Locator

```
class FeedFetcher {
  function __construct() {
    $this->cache =
      Locator::get('cache');
  }
}
```

Configuring With a Locator

```
Locator::set('cache',
    new FileCache('/tmp'));

$fetcher = new FeedFetcher();
```

What is the Difference?

* ```
  Locator::set('cacheDir', '/tmp');
  ...
  $value = Locator::get('cacheDir')
  ```
* ```
  $_GLOBALS['cacheDir'] = '/tmp';
  ...
  $value = $_GLOBALS['cacheDir']
  ```
* ```
  define('CACHE_DIR', '/tmp');
  ...
  $value = CACHE_DIR
  ```
  ```

Service Locator

* Clients have a dependency on the locator
* Difficult to integrate locators from multiple parties
* Easier to use if you are using single instances of an object type
* Easier to use when in deeply nested code
* Less flexible

Drawbacks to Dependency Injection

* All potential dependencies must be instantiated
* Lazy loading requires proxy objects in PHP
* Constructor methods can end up with many parameters

Containers

Dependency Injection

* Separates object configuration from object use

Dependency Injection Container

* Takes over the configuration and assembly of objects
* Uses a domain specific language to describe objects
  * Code Based
  * XML Based

The Value of a Container

* Better for more complicated objects?
* Value added services
* No mature containers for PHP
* I haven't run into a need for one yet

You don't need a Container to Try Dependency Injection

Fluid Interfaces

Standard Configuration API

```
$controller->addParameter(
    new PostParameter('name', 'name'));
```

* What is name?
* Why is it there twice?
* Confusing.
* Most common usage case for this code

Constructor

```
class PostParameter {

  function __contruct(
      $name, $binding = NULL) {
    $this->name = $name;
    $this->bindingLocation = $binding;
  }

}
```

Standard Collection Accessor

```
function addParameter($param) {
  $this->params[] = $param;
}
```

Introduce Non-Standard Collection Accessor

```
function addParameter($param) {
  $this->params[] = $param;
  return $param;
}
```

Move Optional Parameter to Setter

```
class PostParameter {
  function __contruct($name) {
    $this->name = $name;
  }

  function setBinding($binding) {
    $this->bindingLocation = $binding;
  }
}
```

Method Chaining

* Instead of

```
$controller->addParameter(
    new PostParameter('name', 'name'));
```

* We now have

```
$controller->addParameter(
    new PostParameter('name'))
    ->setBinding('name');
```

Adopt Fluid Names

* Instead of

```
$controller->addParameter(
    new PostParameter('name', 'name'));
```

* We now have

```
$controller->DefineInput(
    new PostParameter('name'))
    ->bindToModel('name');
```

Add The Common Default Case

```
class PostParameter {
  function __contruct($name) {
    $this->name = $name;
  }

  function setBinding($binding = NULL) {
    $this->bindingLocation =
      isset($binding) ?
        $binding :
        $name;
  }
}
```

More Understandable?

* Instead of

```
$controller->addParameter(
    new PostParameter('name', 'name'));
```

* We now have

```
$controller->defineInput(
    new PostParameter('name'))
    ->bindToModel();
```

Domain Specific Languages

* Dependency Injection Container
  * Implements a generic DSL for object configuration
* Fluid Interfaces
  * Implements a class specific DSL for configuration

Dependency Injection

* Try it

Questions?