

DESIGN_PATTERNS

Creational Patterns and the Registry Design Pattern

Patrick C. Engel <info@pc-e.org> © 2008

Creational Patterns

- ▾ # 1 Creating a Object
 - ➔ 1.1 Direct Instantiation
 - ➔ 1.2 The Singleton Method
 - ➔ 1.3 The Factory Method

Creational Patterns

- **Direct Instantiation**
using the new Operator

```
$directinstance = new stdClass;
```

Creational Patterns

- ▶ **Singleton:** get a single instance
- ▶ **Singleton Registry:** Implementation to add and get an single instance [with identifier | by key]
- ▶ **Factory:** same task, different implementation. Get Instance of a Class (decides which, where different Classes are possible, same interface)

Goals and Architecture

- ◆ Package and Application Code coupling as necessary
- ◆ generic API
 - ▶ usage of [abstract] interfaces (rather than inheritance)
 - ▶ changing factories is easy, through global singleton access
 - ▶ push dependencies in the object

The Singleton Pattern

- ◆ Singleton
 - ▾ pros
 - one (same) instance
 - performance friendly
 - ▾ drawbacks:
 - alternate instance impossible with new dependencies
(Class that contains Singleton),
 - long (not descriptive) Names to write
`DB::getInstance()->get();`
- ◆ Usage
 - ▾ getInstance method returns an instance of the object
 - ▾ alternative implementations to get another “wanted” Instance

The Singleton Pattern

- ◆ Alternative
 - ▶ Instantiation with Registry Pattern, store an single Instance (the Class is not specifically written as Singleton, but behaves like a Singleton).

```
Zend_Registry::set('db', new Db($cf));  
if(Zend_Registry::isRegistered('db'))  
{  
    $db = Zend_Registry::get('db');  
} else { die ('db Object lost.');
```

The Factory Pattern

- ◆ Factory
 - ▶ the factory object is an abstraction of a constructor, allocated for complex or different Applications
- ◆ Usage:
 - ▶ get concrete Object with descriptive name by client-property, configuration-file or parameterized factory-method

The Abstract Factory Pattern

- ◆ Abstract Factory
 - Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype. [GoF, p95]
- ◆ Usage:

```
$instance = Db  
->getDefaultClass();
```

Development up to present

- ◆ Dependency Injection (**new** is expensive and dangerous, Single Inheritance)
- ◆ IoC (Inversion of Control), SoC
- ◆ modular and re-useable (usage of SPL)
- ◆ Interfaces and abstract Classes
- ◆ keep members private
- ◆ overhead myth
 - ▾ PECL (from PHP as/to C/C++), Cache, persistent connections, **reduce headers[^y!]**, Benchmarks, Kcachegrind, Xdebug (Unit-tests)

Dependency Injection

- ◆ Dependency Injection
 - ▶ Class should not depend on a Configuration
- Bad Example:

```
// $params
$gallery = new Gallery (
    array('path'=>'/img', 'page'=>1)
);

class Gallery {
    public function __construct($params) {
        $images = new GalleryImages($params);
                //      ^           ^
    }
}
```

Dependency Injection

- ◆ Dependency Injection
 - ▶ Class should not depend on a Configuration, push in the dependencies.

```
// DI
$images = new GalleryImages('/img');
$gallery = new Gallery($images);
```

```
class Gallery {
    public function __construct(
        GalleryImages $images) {
        //do something with $images;
    }
}
```

The Registry Pattern

◆ Registry

- ▶ nearly no namespace clash
- ▶ global access through one Storage-Object

◆ USAGE

- ▶ typical usage: static add and get methods
- ▶ alternative: object, with array access etc.

Several Strategies

- ◆ Registry Pattern
 - ▶ Object Instances with Registry
 - ▶ compress data and implement serialize-able methods
 - ▶ building a Register Machine (run and evaluate instructions as object): global vars could be stored in a Registry

Several Strategies

- ◆ Package and Application Code coupling as necessary ,
- ◆ Construction with Factory, Singleton or “Object Pool” with Registry Pattern.

alternative creational Design Pattern

- ◆ Builder pattern
- ◆ Lazy init pattern
- ◆ Object pool pattern
- ◆ Prototype pattern

- ◆ IoC, DI, ServiceLocator don't exclude creational Design Patterns, like Singleton or abstract factory or factory method Design Patterns.

Links

- ▶ this Impression:
<http://code.pc-e.org/php/zend/patterns/>
- ▶ ClassDiagrams
<http://www.cs.ucsb.edu/~mikec/cs50/misc>
- ▶ Factory and/or Singleton[pear::log]:
<http://www.indelible.org/php/Log/guide.ht>
- ▶ Registry Pattern:
<http://framework.zend.com/manual/en/zer>
<http://martinfowler.com/eaCatalog/regist>

Links

- ▶ diagrams:
<http://www.cs.ucsb.edu/~mikeec/cs50/misc>
- ▶ reduce headers:
<http://developer.yahoo.com/performance/>
- ▶ avoid non standard features:
<http://www.ddj.com/cpp/184401958>
- ▶ Improve your code by replacing concrete base classes with interfaces:
<http://www.javaworld.com/javaworld/jw-08>

Links

- ◆ Why extends is evil:
[Interfaces, abstract Classes\[^Java\]](#)
- ◆ avoid non standard features:
[Bjarne Stroustrup Cpp \[^cpp\]](#)
- ◆ rule_of_optimization:
[Prototype before polishing\[^\]](#)