Patterns and OOP in PHP

George Schlossnagle
<george@omniti.com>

What are Patterns?

Patterns catalog solutions to categories of problems

They consist of

- A name
- A description of their problem
- A description of the solution
- An assessment of the pros and cons of the pattern

What do patterns have to do with OOP?

Not so much. Patterns sources outside OOP include:

- Architecture (the originator of the paradigm)
- User Interface Design (wizards, cookie crumbs, tabs)
- Cooking (braising, pickling)

A tasty design pattern

Brining

Problem: Make lean meat jucier

Solution: Submerge the meat in a salt and flavor infused liquid.

Discussion: Salt denatures the meat proteins, allowing them to trap liquid between them more efficiently.

Example: Mix 1C of table salt and 1C of molasses into 1G of water. Bring to a boil to dissolve. Submerge pork roast for 2 days.

What is OOP?

OOP is paradigm more than a feature set.

Everyone is a bit different, and they all think they're right

The classic difference

click(button)

vs.

button.click()

Rephrase: What is its motivation?

Let's try to define OOP through the values it tries to promote.

- Allow compartmentalized refactoring of code
- Promote code reuse
- Promote extensability

Is OOP the only solution for this?

Of course not.

Encapsulation

- Encapsulation is about grouping of related properties and operations into classes.
- Classes represent complex data types and the operations that act on them. An object is a particular instance of a class. For example ‘Dog’ may be a class (it’s a type of thing), while Grendel (my dog) is an instance of that class.

Are Classes Just Dictionaries?

Classses as dictionaries are a common idiom, seen in C:

```
typedef struct _entry {
    time_t date;
    char *data;
    char *(*display)(struct _entry *e);
} entry;
e->display(e);
```

You can see this idiom in Perl and Python, both of which prototype class methods to explicitly grab $this (or their equivalent).

Are Classes Just Dictionaries?

PHP is somewhat different , since PHP functions aren't really first class objects. Still, PHP4 objects were little more than arrays.

The difference is coherency. Classes can be told to automatically execute specific code on object creation and destruction.

```
class Simple {
  function __construct() {/*...*/}
  function __destruct() { /*...*/}
}
```

Leaving a Legacy

A class can specialize (or extend) another class and inherit all its methods, properties and behaviors.

This promotes

- Extensibility
- Reusability
- Code Consolidation

A Simple Inheritance Example

```
class Dog  {
  public function __construct($name) {/*...*/}
  public function bark() { /*...*/ }
  public function sleep() { /*...*/}
  public function eat() { /*...*/}
}
class Rottweiller extends Dog {
  public function intimidate($person);
}
```

Inheritance and the issue of Code Duplication

Code duplication is a major problem for maintainability. You often end up with code that looks like this:

```
function foo_to_xml($foo) {
  // generic stuff
  // foo-specific stuff
}

function bar_to_xml($bar) {
  // generic stuff
  // bar specific stuff
}
```

The Problem of Code Duplication

You could clean that up as follows:

```
function base_to_xml($data) { /*...*/ }
function foo_to_xml($foo) {
  base_to_xml($foo);
  // foo specific stuff
}
```

```
function bar_to_xml($bar) {
  base_to_xml($bar);
  // bar specific stuff
}
```

But it's hard to keep **base_to_xml()** working for the disparate **foo** and **bar** types.

The Problem of Code Duplication

In an OOP style you would create classes for the Foo and Bar classes that extend from a base class that handles common functionality.

```
class Base {
  public function toXML() {  /*...*/ }
}
class Foo extends Base {
  public function toXML() {
    parent::toXML();
    // foo specific stuff
  }
}

class Bar extends Base {
  public function toXML() {
    parent::toXML();
    // Barspecific stuff
  }
}
```

Sharing a base class promotes sameness.

Multiple Inheritance

Multiple inheritance is confusing. If you inherit from ClassA and ClassB, and they both define method foo(), whose should you inherit?

Interfaces allow you to specify the functionality that your class must implement.

Type hints allow you to require (runtime checked) that an object passed to a function implements or inherits certain required facilities.

Multiple Inheritance

```
interface Displayable {
  public function display();
}
class WeblogEntry implements Displayable {
  public function display() { /*...*/}
}
function show_stuff(Displayable $p) {
  $p->display();
}
```

vs.

```
function show_stuff($p) {
  if(method_exists($p, 'display')) {
    $p->display();
  }
}
```

Problem is those checks need to be added in every function.

Abstract Classes

Abstract classes provide you a cross between a 'real' class and an interface. They are classes where certain methods are defined, and other methods are only prototyped.

Abstract classes are useful for providing a base class that should never be instantiated.

```
abstract class CalendarEntry {
  abstract function display();
  public function fetchDetails() { /*...*/}
  public function saveDetails() {/*...*/}
}
```

Public Relations

One of the notions of OOP is that your package/library should have a public API that users should interact with. What happens behind the scenes is none of their business, as long as this public API is stable. This separation is often referred to as 'data hiding' or 'implementation hiding'.

Some languages (Perl, Python) rely on a 'gentleman's contract' to enforce this separation, while other languages enforce it as a language feature.

Data Hiding

PHP implements strict visibility semantics. Data hiding eases refactoring by controlling what other parties can access in your code.

- public anyone can access it
- protected only descendants can access it
- private only you can access it
- final no one can re-declare it.

Why have these in PHP? Because sometimes self-discipline isn't enough.

Minimizing Special Case Handling

Suppose we have a calendar that is a collection of entries. Procedurally dislpaying all the entries might look like:

```
foreach($entries as $entry) {
  switch($entry->type) {
    case 'professional':
      display_professional_entry($entry);
        break;
    case 'personal':
      display_personal_entry($entry); break;
  }
}
```

Simplicity Through Polymorphism

In an OOP paradigm this would look like:

```
foreach($entries as $entry) {
  $entry->display();
}
```

The key point is we don't have to modify this loop to add new types. When we add a new type, that type gets a display() method so it know how to display itself, and we're done.

(p.s. this is a good case for the aggregate pattern, shown later)

Weren't we talking about patterns...?

Singleton Pattern

- Problem: You only want one instance of an object to ever exist at one time
- Solutions:
  - PHP4: Use a factory method with static cache
  - PHP4: Use a global cache and runtime instance mutability
  - PHP5: Use static class attributes

Singleton Pattern

Description:

```
class Singleton {
  static private $instance;
  private function __construct() {}
  static public function instance() {
    if(!self::$instance) {
      self::$instance = new Singleton();
    }
    return self::$instance;
  }
}
$s = Singleton::instance();
```

Static Properties and Methods

Static properties and methods belong to a class as a whole, not a particular instance.

To reference your own static properties, you use:

```
$my_prop = self::$prop;
```

Static properties are not inherited (they are compile-time resolved).To reference a parent's property use:

```
$dads_prop = parent::$prop;
```

To reference an external classes property use:

```
$class_prop = Class::$prop;
```

PPP still applies!

Aggregator Pattern

Problem: You have collections of items that you operate on frequently with lots of repeated code.

Remember our calendars:

```
foreach($entries as $entry) {
  $entry->display();
}
```

Solution: Create a container that implements the same interface, and perfoms the iteration for you.

Aggregator Pattern

```
class EntryAggregate extends Entry {
  protected $entries;
  ...
  public function display() {
    foreach($this->entries as $entry) {
      $entry->display();
  }
  public function add(Entry $e) {
    array_push($this->entries, $e);
  }
}
```

By extending Entry, the aggregate can actually stand in any place that entry did, and can itsefl contain other aggregated collections.

Iterator Pattern

- Problem: You need to be able to iterate through an aggregation.
- Solution: Implement all the operations necessary to logically iterate over an object.

In PHP this is now a built in facility so that object implementing the Iterator interface can be used in the language array constructs like foreach().

An Infinite Iterator

```
class EndlessSquares implements
Iterator {
  private $idx = 0;
  function key() {
    return $this->idx;
  }
  function current() {
    return $this->idx * $this->idx;
  }
  function next() {
    $this->idx++;
    return $this;
  }
  function valid() {
    return true;
  }
  function rewind() {
    $this->idx = 0;
  }
}
```

```
es = new EndlessSquares;
foreach($es as $square) {
    echo "$sq\n";
}
```

Aren't Iterators Pointless in PHP?

Why not just use:

```
foreach($aggregate as $item) { /*...*/ }
```

Aren't we making life more difficult than need be?

No! For simple aggregations the above works fine, but not everything is an array. What about:

- Buffered result sets
- Directories
- Anything not already an array

Proxy Pattern

- Problem: You need to provide access to an object, but it has an interface you don't know at compile time.
- Solution: Use accessor/method overloading to dynamically dispatch methods to the object.
- Discussion: This is very typical of RPC-type facilities like SOAP where you can interface with the service by reading in a definitions file of some sort at runtime.

Proxy Pattern in PEAR SOAP

```
class SOAP_Client {
    public $wsdl;
    public function __construct($endpoint) {
        $this->wsdl = WSDLManager::get($endpoint);
    }
    public function __call($method, $args) {
        $port = $this->wsdl->getPortForOperation($method);
        $this->endpoint = $this->wsdl->getPortEndpoint($port);
        $request = SOAP_Envelope::request($this->wsdl);
        $request->addMethod($method, $args);
        $data = $request->saveXML();
        return SOAP_Envelope::parse($this->endpoint, $data);
    }
}
```

Observer Pattern

- Problem: You want an object to automatically notify dependents when it is updated.
- Solution: Allow 'observer' to register themselves with the observable object.
- Discussion: An object may not apriori know who might be interested in it. The Observer pattern allows objects to register their interest and supply a notification method.

Observer Pattern

```
class Observable {
  protected $observers;
  public function attach(Observer $o) {
    array_push($this->observers, $o);
  }
  public function notify() {
    foreach($this->observers as $o) {
      $o->update();
    }
  }
}
interface Observer {
  public function update();
}
```

Concrete Examples: logging facilities: email, debugging, SOAP message notifications. NOT Apache request hooks.

THANKS!

- Slides for this talk will be available shortly at
  http://www.omniti.com/~george/talks/
- A longer version with 3 hours of info and a greater focus on patterns and advanced features will be presented by Marcus Boerger and myself at OSCON in July. Come see us there!
- Shameless book plug: Buy my book, you'll like it. I promise.