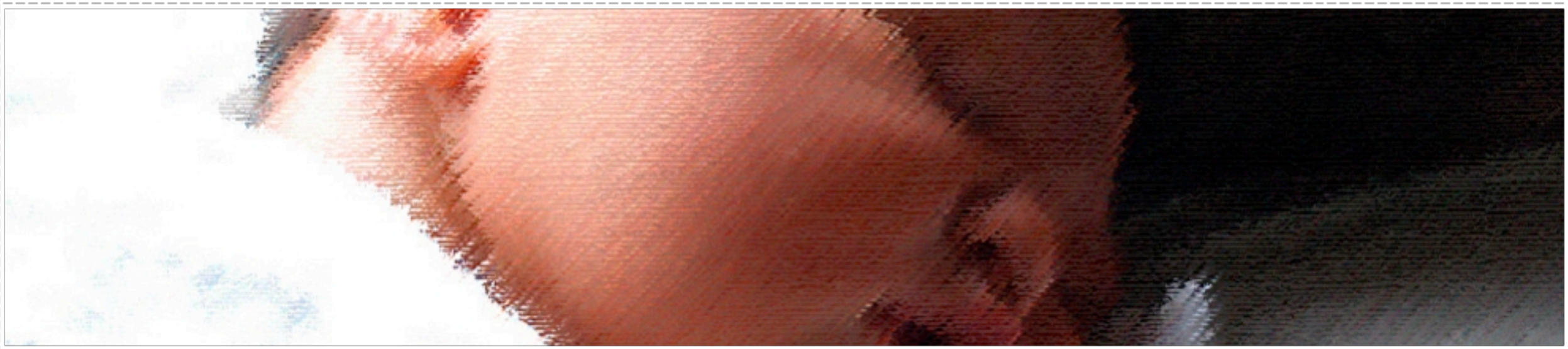
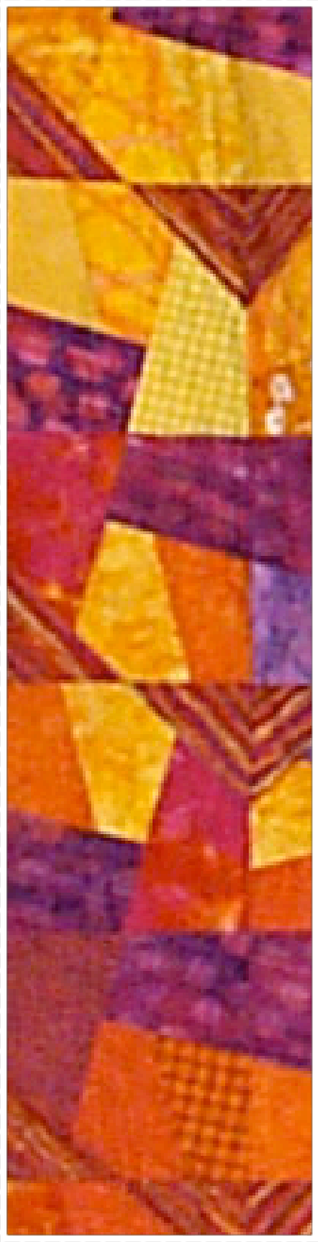
Patterns and OOP in PHP

George Schlossnagle
<george@omniti.com>

Patterns (I)



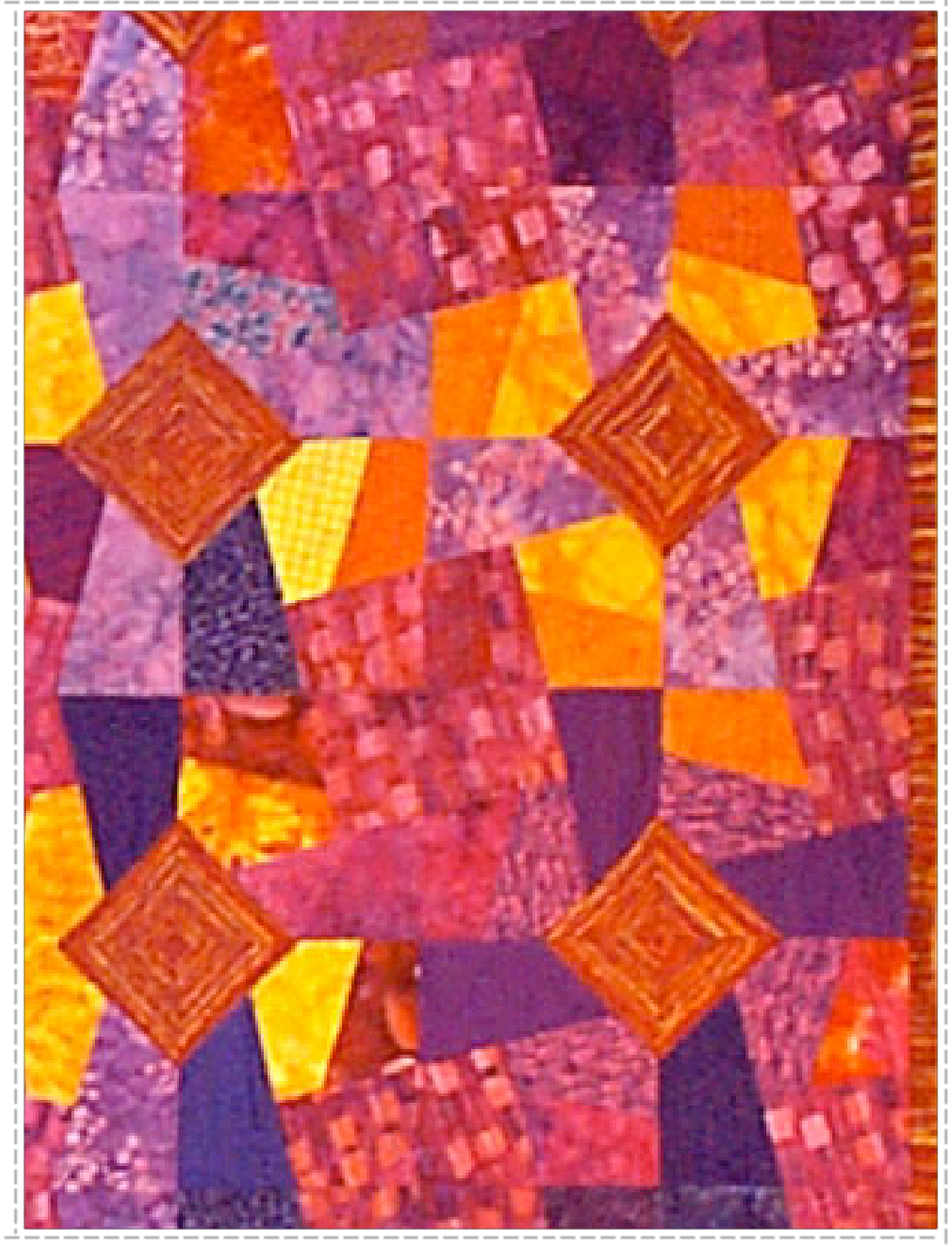
What are Patterns?



"Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

—Christopher Alexander, The Timeless Way of Building, 1979

So what are patterns?



What are Patterns, really?

Patterns catalog solutions to categories of problems

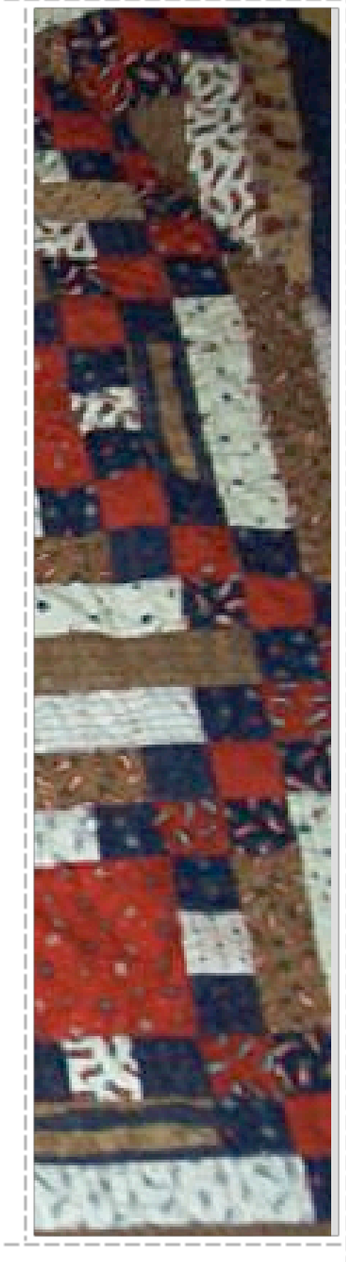
They consist of

A name

A solution to a problem

A description of the solution

An assessment of the pros and cons of the pattern



What are Patterns Not?

- An algorithm.
- An implementation.
- The same every time you see them.

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)

Brining: A tasty design pattern

- 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.

A non-OO example: Authentication

Problem: You need to provide session authentication.

Solution: Use a secret that the user presents you to identify the user.

Discussion: A reference example uses a cookie to hold the secret. The cookie is transmitted on each request to your site. The secret can be used to verify the users identity.

Why are patterns associated with OOP?

Brought into the industry in a time when OOP was über-hot.

Reference implementations become more useful when they can be derived from directly.

Patterns are Useful

I have a DB wrapper class I use frequently

```
class DB_Mysql {
  protected $user;
  protected $pass;
  protected $dbhost;
  protected $dbname;
  protected $dbh;

  function __construct($user, $pass, $dbhost, $dbname) {
    $this->user = $user;
    $this->pass = $pass;
    $this->dbhost = $dbhost;
    $this->dbname = $dbname;
  }
  function connect() { //...
```

I don’t like having to pass in the connection parameters at instatiation time, and I would like to be able to globally set options for a given database (for example, to enable debugging).

Patterns are Useful

The Solution: The Template Pattern

Discussion: The template pattern uses a base class as a template for a series of children, each of whom will supply the necessary information to complete it.

My implementation:

```
class DB_Mysql_Foo extends DB_Mysql {
    protected $user   = "foo";
    protected $pass   = "bar";
    protected $dbhost = "10.1.1.3";
    protected $dbname = "Foo";

    function __construct() {
      if(strncmp($_ENV['HOSTNAME'], 'devel-1', strlen('devel-1')) == 0) {
        $this->dbhost = '127.0.0.1:3308';
      }
    }
}
```

Patterns are Useful

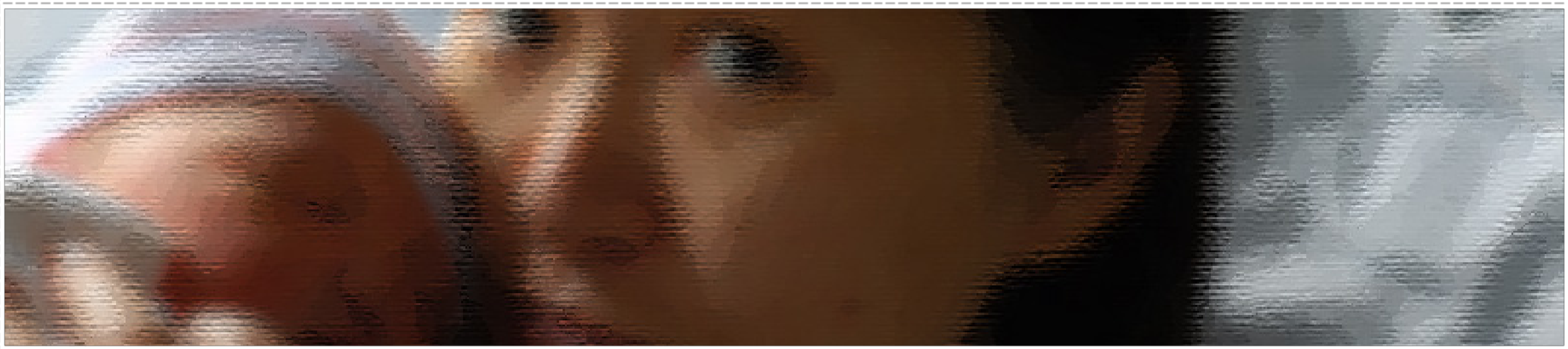
Now, whenever I need to make a connection to the **Foo** database, I only need to do:

```
$dbh = new Foo;
```

In addition to looking very clean:

- Connection parameters are hidden from the user.
- Debugging can be enabled by overloading individual functions.

Intro to OOP



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($p) {
  if(is_object($p) && method_exists($p, 'display')) {
    $p->display();
  }
}
```

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

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, or for abstract (incomplete) pattern implementations.

```
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();
}
```

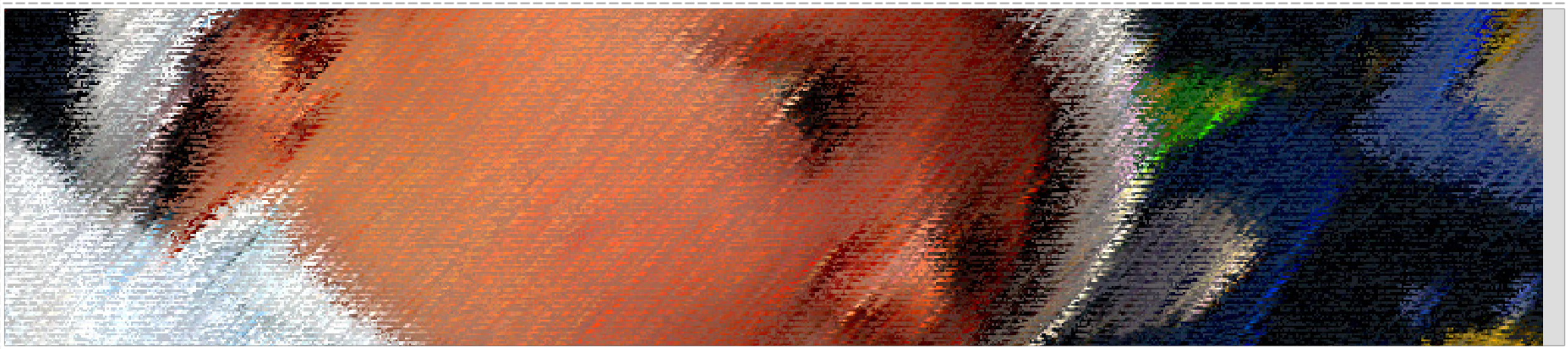
The key point is we don't have to modify this loop to add new types. When we add a new type, that type gets a display() method so it know how to display itself, and we're done.

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

The PHP5 OOP Feature List

- Objects are referenced by identifiers
- Constructors and Destructors
- Static members
- Default property values
- Constants
- Visibility
- Interfaces
- Final and abstract members
- Interceptors
- Exceptions
- Reflection API
- Iterators

Patterns by Example



Singleton

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: You need a class that can only have a single instance at any time.

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

Singleton in Use

```
$s1 = Singleton::getInstance();
$s2 = Singleton::getInstance();

$s1->foo = 'bar';
echo $s2->foo; // prints "bar"
```

Factory

Factory Pattern

- The Factory Pattern lets you create instances of multiple classes based on runtime input.

Factory Pattern Applied

A calendaring app needs to support arbitrary item types.

Items have unified IDs, so when you fetch an item, you don't apriori know what type it is.

```
class CalendarItem {
  public function getCalendarItemByID($id)
  {
    $dbh = new DB_Calendar;
    $sth = $dbh->execute("SELECT type
                          FROM calendar_items
                          WHERE id = $id");
    if($data = $sth->fetch_assoc()) {
      $item = new $data['type']($data['id']);
    }
  }
  protected __construct() {}
}

class RecurringEvent extends CalendarItem {
  public function __contruct($id)
  {
    //  class-specific constructor
    parent::__construct();
  }
}
```

Factory in Use

```
$sth = $dbh->query("SELECT id from calendar_items
          WHERE start_time < date_add(now(), '2 weeks'))";
while(list($id) = $sth->fetch()) {
    $items[] = CalendarItem::getCalendarItemByID($id);
}
```

Aggregate Pattern

Aggregate Pattern

You have collections of items that you operate on frequently with lots of repeated code. The Aggregate Pattern gives you an efficient manner for dealing with collections.

Remember our calendars:

```
foreach($items as $item) {
  $item->display();
}
```

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

Aggregator Pattern

```
class CalendarAggregate extends CalendarItem {
  protected $items;
  public function add(CalendarItem $item) {
    $items[] = $item;
  }
  public function display() {
    foreach($this->items as $item) {
      $item->display();
    }
}
```

By extending CalendarItem, the aggregate can actually stand in any place that did, and can itself contain other aggregated collections.

Aggregate in Use

```
$sth = $dbh->query("SELECT id from calendar_items
        WHERE start_time < date_add(now(), '2 weeks'))";
$cal = new CalendarAggregate;
while(list($id) = $sth->fetch()) {
    $cal->add(CalendarItem::getCalendarItemByID($id));
}

// ...

$cal->display();
```

Iterator

Iterator Pattern

The Iterator Pattern provides a way to iterate through an object in a proscribed fashion.

Aren’t Iterators Pointless in PHP?

Why not just collect your items in an array and use:

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

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

No! For simple aggregations the above works fine (though it’s slow), but not everything is an array. What about:

- Values and Keys in an array
- Text lines in a file
- Database query results
- Files in a directory
- Elements or Attributes in XML
- Bits in an image
- Dates in a calendar range

Reading an INI File

```
class DbaReader implements Iterator {
  protected $db = NULL;
  private $key = false, $val = false;

  function __construct($file, $handler) {
    if (!$this->db = dba_open($file, 'r', $handler))
      throw new exception('Could not open file ' . $file);
  }
  function __destruct() { dba_close($this->db); }
  function rewind() {
    $this->key = dba_firstkey($this->db);
    $this->fetch_data();
  }
  function next() {
    $this->key = dba_nextkey($this->db);
    $this->fetch_data();
  }
  private function fetch_data() {
    if ($this->key!==false)
    $this->val = dba_fetch($this->key, $this->db);
  }
  function current() { return $this->val; }
  function valid() { return $this->key !== false; }
  function key() { return $this->key; }
}
```

Reading an INI File

```
$d = new DbaReader("/etc/php/php.ini", "inifile");
foreach($d as $k => $v) {
  print "$k => $v\n";
}
```

The Problem with the INI Reader

Its output contains all the commented entries.

```
[PHP]enable_dl => On
[PHP]; cgi.force_redirect => 1
[PHP]; cgi.nph => 1
[PHP]; cgi.redirect_status_env => ;
[PHP]; fastcgi.impersonate => 1;
[PHP];cgi.rfc2616_headers => 0
```

That’s not terribly useful.

Decorator

Decorator

The Decorator pattern provides a way to add additional functionality to an object at runtime. The Decorator object wraps the object to be decorated, proxying certain calls and handling others on its own.

FilterIterator

FilterIterator is a builtin Iterator class that uses the Decorator pattern to wrap another iterator and returns only select elements of the inner iterator. **FilterIterator** is implemented in C, but if it was in PHP it would look like it does on the right.

Notice here how all the calls are proxied except **next()**.

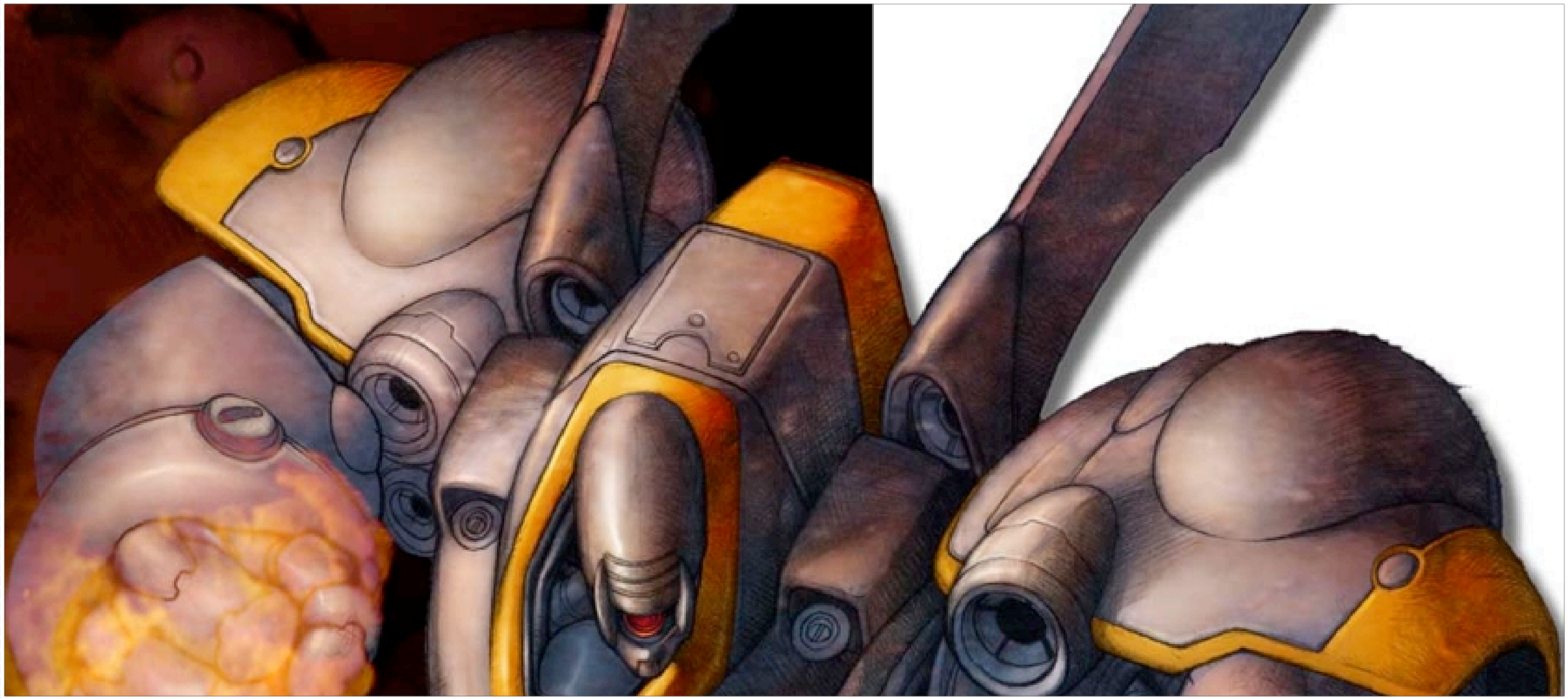
```
abstract class FilterIterator implements Iterator {
    private $iterator;
    function __construct(Iterator $iterator) {
        $this->iterator = $iterator;
    }
    abstract function accept();
    function getInnerIterator() { return $this->iterator; }
    function rewind() { return $this->iterator->rewind(); }
    function key() { return $this->iterator->key(); }
    function current() { return $this->iterator->current(); }
    function valid() { return $this->iterator->valid(); }
    function next() {
        $next;
        do {
            $this->iterator->next();
        while($this->accept() == 0)
    }
}
```

IniFilter

```
class IniFilter extends FilterIterator {
  function accept() {
    $val = preg_replace("/^\[[^]]*]/", "",
            $this->getInnerIterator()->key());
    $val = preg_replace("/[;#].*/", "", $val);
    return $val?1:0;
  }
}

$d = new DbaReader("/etc/php/php.ini", "inifile");
$filter = new  IniFilter($d);

foreach($filter as $k => $v) {
  print "$k => $v\n";
}
```



Proxy

Proxy Pattern

Problem: You need to provide access to an object, but it has an interface you don't know at compile time. 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);
    }
}
```

SOAP in Use

```
$soap = new SOAP_Client("http://www.example.com/service.wsdl");
$soap->SomeFunction($data);
```

Observer

Observer Pattern

The Observer pattern shows a flexible way for objects to notify interested parties on certain events.

The Problem wit PEAR_Error

```
if ($this->mode & PEAR_ERROR_TRIGGER) {
    trigger_error($this->getMessage(), $this->level);
}
if ($this->mode & PEAR_ERROR_DIE) {
    $msg = $this->getMessage();
    if (is_null($options) || is_int($options)) {
        $format = "%s";
        if (substr($msg, -1) != "\n") {
            $msg .= "\n";
        }
    } else {
        $format = $options;
    }
    die(sprintf($format, $msg));
}
if ($this->mode & PEAR_ERROR_CALLBACK) {
    if (is_callable($this->callback)) {
        call_user_func($this->callback, $this);
    }
}
if ($this->mode & PEAR_ERROR_EXCEPTION) {
    trigger_error("PEAR_ERROR_EXCEPTION is obsolete, use class PEAR_ErrorStack for exceptions", E_USER_WARNING);
    eval('$e = new Exception($this->message, $this->code);$e->PEAR_Error = $this;throw($e);');
}
```

What happened to single responsibility?

A More Flexible Solution

Let handlers be registered and called out to on error:

```
interface ErrorObserver {
  function update(PEAR_Error $e);
}
class PEAR_Error {
  private static $handlers;
  function addHandler(ErrorObserver $h) {
    self::$handlers[] = $h;
  }
  function raiseError() {
    foreach(self::$handlers as $h) {
      $h->update($this);
    }
  }
}
```

Advantages

The core error class doesn't need to be modified if you want to add a new possible error reaction.

Custom handlers are very easy:

```
class MailError implements ErrorObserver {
  private $to;
  function __construct($to) {
    $this->to = $to;
  }
  function update(PEAR_Error $e) {
    mail($this->to, "Error Occured", $e->getMessage());
  }
}
PEAR_Error::addHandler(new MailError('george@example.com'));
```

Serializer

Serializer

The serializable pattern describes a standard way of marshalling and unmarshalling object data.

Unserializer

PHP provides native support through this pattern via **__sleep()**, **__wakeup()**, and **unserialize_callback_func**.

Serializer in Use

```
class DB_Mysql {
  function __sleep() {
     unset($this->dbh);
  }
  function __wakeup() {
    $this->connect();
  }
}

function _wakeup_helper($class) {
  include_once("$class.inc");
}
ini_set('unserialize_callback_func', '_wakeup_helper');
```

THANKS!

Slides for this talk will be available shortly at http://www.omniti.com/~george/talks/

Shameless book plug: Buy my book, you'll like it. I promise.