Designing Beautiful Software

Matthew Weier O’Phinney

26 May 2011

What is Beauty?

What is Beauty?

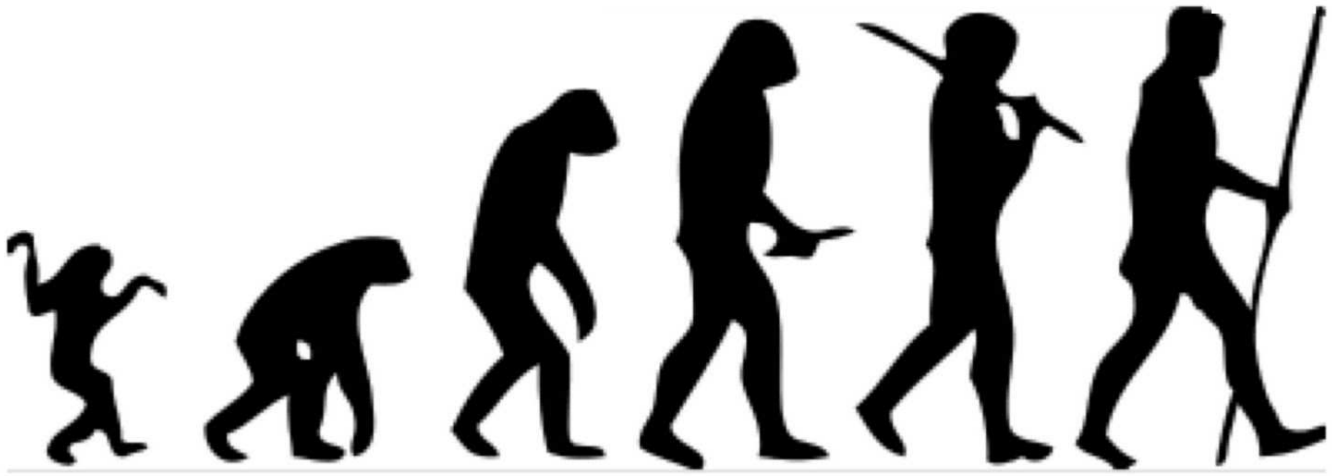
What is beautiful software design?

What is beautiful software design?

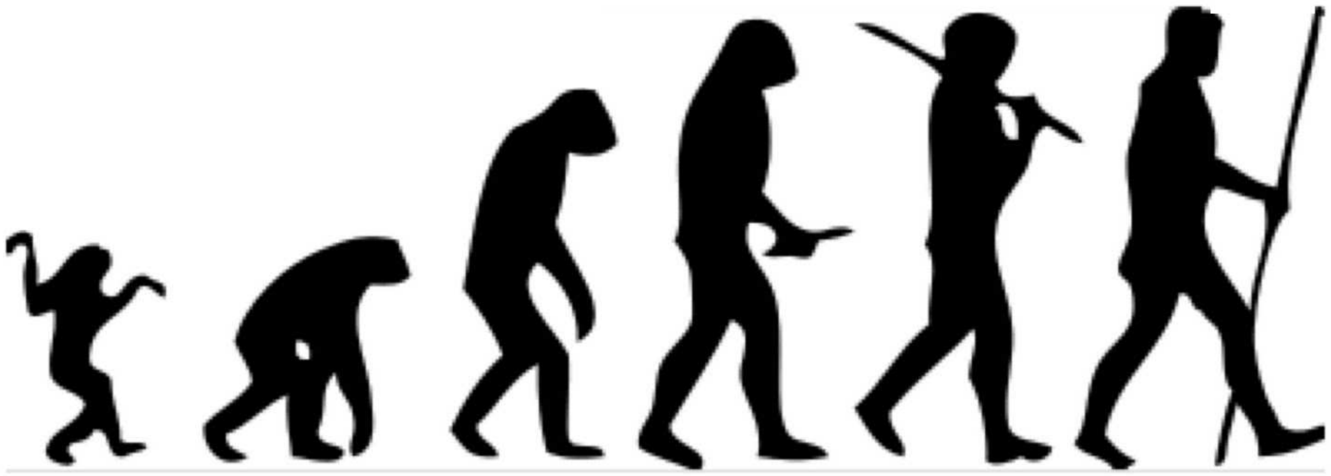
What is beautiful software design?

What is beautiful software design?

Software Evolution



Software Evolution



or, how to go from code monkey to architect

Examining a real-world problem

Requirements

- The application needs to send email

Requirements

- The application needs to send email
- I know the sender is always the same

Requirements

- The application needs to send email
- I know the sender is always the same
- I want to BCC an address for verification

```php
function shop_mail($to, $subject, $body)
{
    $headers = "From: shop@example.com\r\n"
             .= "Bcc: shop-sent@example.com\r\n";
    mail($to, $subject, $body, $headers);
}
```

Is it beautiful?

- It’s succinct

Is it beautiful?

- It’s succinct
- It prevents us having to specify `$headers` manually each call

Is it beautiful?

- It’s succinct
- It prevents us having to specify `$headers` manually each call
- It’s little more than a wrapper on `mail()`

What if we introduce requirements?

- For instance, we add another shop on a different domain, using much (if not all) the same code.

What if we introduce requirements?

- For instance, we add another shop on a different domain, using much (if not all) the same code.
- Now the “From” and “Bcc” addresses need to be different.

New problems

- We don’t want to change all the places in our code that call this function…

New problems

- We don’t want to change all the places in our code that call this function. . .
- . . . at least, not after this change.

New problems

- We don’t want to change all the places in our code that call this function…
- …at least, not after this change.
- So, let’s introduce a “configuration” parameter.

```php
function shop_mail($to, $subject, $body, $shop = 'original')
{
    switch ($shop) {
        case: 'subdomain':
            $from = 'shop@subdomain.example.com';
            $bcc  = 'shop-sent@subdomain.example.com';
            break;
        case: 'original':
        default:
            $from = 'shop@example.com';
            $bcc  = 'shop-sent@example.com';
            break;
    }

    $headers = "From: $from\r\n"
            .= "Bcc: $bcc\r\n";
    mail($to, $subject, $body, $headers);
}
```

Is it beautiful?

- Switch statements will grow, and need to be documented.

Is it beautiful?

- Switch statements will grow, and need to be documented.
- We need to know what the value of that last argument will be.

Is it beautiful?

- Switch statements will grow, and need to be documented.
- We need to know what the value of that last argument will be.
- The number of arguments may not justify wrapping `mail()`

Evolution:
use classes

Base functionality

```php
class ShopMail
{
    protected static $from = 'shop@example.com';
    protected static $bcc  = 'shop-sent@example.com';

    public static function send($to, $subject, $body)
    {
        $headers = "From: " . static::$from . "\r\n"
                .= "Bcc: " . static::$bcc . "\r\n";
        mail($to, $subject, $body, $headers);
    }
}
```

Subclass

```php
class SubdomainMail extends ShopMail
{
    protected static $from = 'shop@subdomain.example.com';
    protected static $bcc  = 'shop-sent@subdomain.example.com';
}
```

Potential usage

```
define('MYENV', 'Subdomain');

$mailer = MYENV . 'Mail::send';
call_user_func($mailer, $to, $subject, $body);
```

Is it beautiful?

- Requires extension

Is it beautiful?

- Requires extension
- Strategy selection requires knowledge of environment

Is it beautiful?

- Requires extension
- Strategy selection requires knowledge of environment
- Debugging requires knowledge of environment

Evolution:
use configuration

```
$config = new ArrayObject(array(),
    ArrayObject::ARRAY_AS_PROPS);
$config->env = "Subdomain';

$mailer = $config->env . 'Mail::send';
call_user_func($mailer, $to, $subject, $body);
```

Is it beautiful?

- Would be easier to just indicate the class to use, and have it be the same throughout the application(s).

Is it beautiful?

- Would be easier to just indicate the class to use, and have it be the same throughout the application(s).
- What if I have a new requirement, such as sending HTML mails?

Evolution:
use *objects*

Mailer class

```
class Mailer
{
    protected $from = 'shop@example.com';
    protected $bcc  = 'shop-sent@example.com';
    protected $contentType = 'text/plain';

    public function setFrom($from) { ... }
    public function setBcc($bcc) { ... }
    public function setContentType($type) { ... }
    public function send($to, $subject, $body)
    {
        $headers = "From: " . $this->from . "\r\n"
                .= "Bcc: " . $this->bcc . "\r\n"
                .= "Content-Type: " . $this->contentType .
                    "\r\n";
        mail($to, $subject, $body, $headers);
    }
}
```

Usage

```
$mailer = new Mailer();
$mailer->setFrom($config->from)
       ->setBcc($config->bcc)
       ->setContentType('text/html');
$mailer->send($to, $subject, $body);
```

Is it beautiful?

- Better, but not great

Is it beautiful?

- Better, but not great
- Specific headers are hard-coded

Is it beautiful?

- Better, but not great
- Specific headers are hard-coded
- What if we don't want to use `mail()`?

Take a shower

- We’ve identified several needs:

- We’ve identified several needs:
  - Configurable, arbitrary headers
  - Configurable, arbitrary transports

Evolution:
use *composition* and *interfaces*

Mail transports and headers

```
interface MailTransport
{
    public function send($to, $subject, $body, $headers);
}

class MailHeaders extends ArrayObject
{
    public function toString()
    {
        $headers = '';
        foreach ($this as $header => $value) {
            $headers .= $header . ': ' . $value . "\r\n";
        }
        return $headers;
    }
}
```

Mailer: setters

```
class Mailer
{
    protected $headers, $transport;

    public function setHeaders(MailHeaders $headers) {
        $this->headers = $headers;
        return $this;
    }

    public function getHeaders() {
        return $this->headers;
    }

    public function setTransport(MailTransport $transport) {
        $this->transport = $transport;
        return $this;
    }

    /* ... */
}
```

Mailer: construction and functionality

```
class Mailer
{
    protected $headers, $transport;

    public function __construct(MailTransport $transport)
    {
        $this->setTransport($transport);
        $this->setHeaders(new MailHeaders());
    }
    public function send($to, $subject, $body)
    {
        $this->transport->send(
            $to, $subject, $body,
            $this->headers->toString()
        );
    }
}
```

Mailer: usage

```
$mailer  = new Mailer(new SmtpTransport);
$headers = new MailHeaders();
// or $headers = $mailer->getHeaders();

$headers['From']         = $config->from;
$headers['Bcc']          = $config->bcc;
$headers['Content-Type'] = 'text/html';
$mailer->setHeaders($headers) // if instantiated separately
       ->send($to, $subject, $body);
```

Is it beautiful?

- Headers management and serialization is self-contained

Is it beautiful?

- Headers management and serialization is self-contained
- Sending is separate from message composition

Is it beautiful?

- Headers management and serialization is self-contained
- Sending is separate from message composition
- Problems
  - No validation or normalization of header keys

Is it beautiful?

- Headers management and serialization is self-contained
- Sending is separate from message composition
- Problems
  - No validation or normalization of header keys
  - No validation of header values

Is it beautiful?

- Headers management and serialization is self-contained
- Sending is separate from message composition
- Problems
  - No validation or normalization of header keys
  - No validation of header values
  - Should a message send itself? or should we pass a message to the transport?

Evolution:
semantic object relations

Message

```
interface MailMessage
{
    public function setTo($to);
    public function setSubject($subject);
    public function setBody($body);
    public function setHeaders(MailHeaders $headers);

    public function getTo();
    public function getSubject();
    public function getBody();
    public function getHeaders();
}
```

Headers and Transport

```
interface MailHeaders
{
    public function addHeader($header, $value);
    public function toString();
}

interface MailTransport
{
    public function send(MailMessage $message);
}
```

Usage

```
$message = new Message();
$headers = new MessageHeaders();
$headers->addHeader('From', $from)
        ->addHeader('Content-Type', 'text/html');
$message->setTo($to)
        ->setSubject($subject)
        ->setBody($body)
        ->setHeaders($headers);
$transport = new SmtpTransport($config->transport);
$transport->send($message);
```

Is it beautiful?

- Headers can now potentially have validation and normalization…

Is it beautiful?

- Headers can now potentially have validation and normalization. . .
  - . . . and implementation can be varied if necessary.

Is it beautiful?

- Headers can now potentially have validation and normalization…
  - …and implementation can be varied if necessary.
- The message is self-contained, and contains all metadata related to it.

Is it beautiful?

- Headers can now potentially have validation and normalization…
  - …and implementation can be varied if necessary.
- The message is self-contained, and contains all metadata related to it.
- The mail transport accepts a message, and determines what to use from it, and how.

Is it beautiful?

What are the real questions?

- Can I maintain it?

What are the real questions?

- Can I maintain it?
- Can I change how it works easily as needed?

Other considerations

- Adopt a coding standard.

Other considerations

- Adopt a coding standard.
- Adopt a sane class -> filesystem convention.

Other considerations

- Adopt a coding standard.
- Adopt a sane class -> filesystem convention.
- Think about how classes relate semantically, and apply this to the class hierarchy.

Other considerations

- Adopt a coding standard.
- Adopt a sane class -> filesystem convention.
- Think about how classes relate semantically, and apply this to the class hierarchy.
  - Consider how this relates to namespaces.

Test first

- Define your requirements as tests.

Test first

- Define your requirements as tests.
- Play with the API and how you use the code before you write it.

Test first

- Define your requirements as tests.
- Play with the API and how you use the code before you write it.
- Having tests ensures that as you fix bugs or introduce features, you don’t break your original contract.

Trade offs

You can always write shorter code. It just likely won’t be as configurable or extensible.

Trade offs

You can always write shorter code. It just likely won't be as configurable or extensible.

- There's nothing *wrong* with the following code. It's fast, and easily understandable.

```
$headers = "From: " . $config->from . "\r\n"
        .= "Bcc: " . $config->bcc . "\r\n";
mail($to, $subject, $body, $headers);
```

Trade offs

You can always write shorter code. It just likely won't be as configurable or extensible.

- There's nothing *wrong* with the following code. It's fast, and easily understandable.

```
$headers = "From: " . $config->from . "\r\n"
        .= "Bcc: " . $config->bcc . "\r\n";
mail($to, $subject, $body, $headers);
```

- However, we can't swap out the transport easily, or test it.

Trade offs

Writing configurable or extensible code usually requires *some* verbosity.

Trade offs

Writing configurable or extensible code usually requires *some* verbosity.

- Separating out objects by areas of concern leads to a proliferation of objects.

Trade offs

Writing configurable or extensible code usually requires *some* verbosity.

- Separating out objects by areas of concern leads to a proliferation of objects.
- Deal with it.

Trade offs

Writing configurable or extensible code usually requires *some* verbosity.

- Separating out objects by areas of concern leads to a proliferation of objects.
- Deal with it.
- It’s easier to digest small bites than it is a whole roast at a time.

Trade offs

Configuration can be either inline, or from a container.

Trade offs

Configuration can be either inline, or from a container.

- Inline is nice, but makes it difficult to swap out later.

Trade offs

Configuration can be either inline, or from a container.

- Inline is nice, but makes it difficult to swap out later.
- Containers are nice, but you then need to pass the container around somehow.

Choose beauty

- Think about long-term maintainability
- Think about extensibility

Thank you!

- Feedback: `http://joind.in/3395`
- Twitter `http://twitter.com/weierophinney`