Pisząc poradę o wysyłaniu emaili od razu chciałem zrobić klasę zabezpieczająca przed wszystkim. Jednak w pewnym momencie zauważyłem, że kod ma około 100 linii, a miała być to prosta porada. Dlatego wróciłem do możliwie najprostszego kodu, a tu mam zamiar trochę rozwinąć swoją myśl. Na tyle dużo, aby pokazać, o co mi chodzi, ale nie a tyle aby zrobić to za ciebie :).

Główne wady poprzedniego kodu:

- brak walidacji danych wejściowych
- brak informacji dlaczego mail się nie wysłał (nieprawidłowe dane, błędne działanie funkcji mail)
- tylko jeden odbiorca

Już w tamtej poradzie wspomniałem, że celowo kod jest trochę "nadmiarowy". Teraz postaram się odpowiedzieć na pytanie, po co jest nadmiarowy.

Walidacja danych wejściowych

Zróbmy sobie tablicę pozwalającą nam walidować dane wejściwoe wedle uznania (wykorzystując wyrażenia regularne).

{codecitation class="brush: php;"}class Mailer {
    private $validation = array('to' => '/@/i', 'subject' => '/.{1,100}/');
    private $to   = false; // do kogo wyslac {/codecitation}

Celowo dałem linię przed i linię po, abyś wiedział gdzie ową zmianę w kodzie umieścić.

Jak widać, chcę sprawdzać jedynie email oraz tytuł. Wyrażenia regularne są tu bardzo ubogie. Celowo - nie chodzi tu o naukę regExpów, ale o pewne podejście do projektu. Zauważ, że w

tablicy yMailer::validation nie ma pola 'content'. Zakładam, że content nie musi być validowany.

Skoro już mamy tablicę walidacyjną, to jeszcze by się przydało ją jakoś wykorzystać. Ale w którym momencie? Ja proponuję, aby zrobić to przy dodawaniu danych.

{codecitation class="brush: php;"}public function recipient($to) {

```
    if ($this->validInput($to, 'to')) {

        $this->to = $to; // przeslo test - zapiszmy wartosc

        return true; //  udało się ustawić odbiorcę.

    }

    return false; // nie udało się ustawić odbiorcy

}
```

```
public function subject( $subject ) {

    if ($this->validInput($subject, 'subject')) {
```

```
        $this->subject = $subject; // przeslo test - zapiszmy wartosc

        return true; //  udało się ustawić tytul.

    }

return false; // nie udało się ustawić tytulu

}

public function content( $content ) {

    if ($this->validInput($content, 'content')) {

        $this->content = $content; // przeslo test - zapiszmy wartosc

        return true; //  udało się ustawić tresc.

    }

return false; // nie udało się ustawić tresc
```

}{/codecitation}

W powyższym kodzie jeszcze jedna rzecz jest niejasna. Czym u licha jest yMailer::validInput?! Już dopisuję:

{codecitation class="brush: php;"} private $content = false; // ta linia tu jest tylko po to, abys wiedzial gdzie znajduje sie ten kod

private function validInput($input, $name) {

if (isset($this->validation[$name])) {// to pole jest ustawione w $validation => ma być sprawdzane

if (preg_match($this->validation[$name], $input)) {

return true; // przeszlo test z wyrazeniem regularnym

} else {

return false;

}

```
        } else {

            return true; // skoro nie trzeba tego sprawdzac, to zawsze jest prawidlowe

        }

    }
```

```
    public function recipient( $to ) { // ta linia tu jest tylko po to, aby s wiedzial gdzie ten kod się znajduje{/codecitation}
```

No, teraz już mamy walidację.Nadal jednak coś mi się nie podoba. Co jeśli chcę mieć inne wyrażenia regularne do sprawdzania podanych wartości? Może zrobić yMailer::validation publiczną? Byłaby możliwość ustawiania sobie dowolnej zmiennej... Nie. To by nam znowu związało interfejs z implementacją, a dążymy do tego, aby interfejs pozostawał taki sam. Nawet jeśli wewnątrz klasa działa zupełnie inaczej.

Proponuję zrobić tak:

```
{codecitation class=&amp;quot;brush: php;&amp;quot;}    private $content = false; // tresc wiadomosci

   public function __construct( $validation = null ) {
```

```
        if (is_array($validation)) {

            $this->validation = $validation;

        }

    }

    private function validInput( $input, $name ) {{/codecitation}
```

Teraz możemy tworzyć obiektyu na dwa sposoby:

{codecitation class="brush: php;"}$mail1 = new yMailer(array('content' => '/wow/')); // chcemy aby w tresci bylo wow. Nie interesuje nas wcale email i tytul

$mail2 = new yMailer(); // polegamy na domyslnych wyrazeniach regularnych{/codecitation}

jak widzisz, teraz możemy sami decydować co chcemy sprawdzać i według jakich zasad. Oczywiście jest tu przydatna znajomość wyrażeń regularnych... Jednak w internecie można znaleźć wiele miejsc, gdzie można się ich poduczyć, lub znaleźć gotowce.

Kolejnym punktem w wadach powyższej klasy było 'brak informacji o tym, 'dlaczego mail nie został wysłany'.

Ustalmy 3 scenariusze:

- porażka z powodu nieprawidłowych danych (np. wywołanie yMailer::send przed ustawieniem wszystkich danych)
- porażka z powodu czegoś innego (ale dane same w sobie wypełnione)
- sukces - funkcja mail zwraca prawdę

Jak widać są trzy scenariusze. A więc true / false to za mało. Zrobimy -1 / 0 / 1.

Oto nowy kod metody yMailer::send():

{codecitation class="brush: php;"} public function send() {

$result = -1; // zakladam, ze jest cos nie tak z danymi

if (false === $this->to or

false === $this->subject or

```
        false === $this->content) {

        if (mail($this->to, $this->subject, $this->content)) {

            $result = 1; // udalo sie wyslac

        } else {

            $result = 0; // nie udalo sie wyslac, ale dane byly wypelnione

        }

    }
  return $result; // zwracam flagę

}{/codecitation}
```

Dzięki takiemu zabiegowi mogę teraz stwierdzić, czy nie udało się wysłać maila z powodu nieustawienia jakichś danych, czy też wina leży w czym innym.

Refaktoryzacja kodu

Zawsze po napisaniu i przetestowaniu kodu sprawdź, czy nie da się go uprościć. Ja widzę w powyższym dwa miejsca, które chciałbym uprościć. Pierwsze to powtarzający się kod metod yMailer::recipient(), yMailer::content() i yMailer::subject(). Zobaczmy czy coś się da z tym zrobić...

{codecitation class="brush: php;"}

```
    private function add( $value, $name ) {
        if ($this->validInput($value, $name)) {
            $this->$name = $value; // przeslo test - zapiszmy wartosc
        return true; //  udało się ustawić odbiorcę.
        }

    return false; // nie udało się ustawić odbiorcy
    }

    public function recipient( $to ) {
        return $this->add($to, 'to');
    }

    public function subject( $subject ) {
        return $this->add($subject, 'subject');
    }

    public function content( $content ) {
        return $this->add($content, 'content');
    }
```

{/codecitation}

Od razu trochę czytelniej. Co więcej musimy sprawdzić jedynie czy działa metoda yMailer::add(), a nie testować 3 inne. Zmusza nas to dodatkwo do schamtyczności w kodzie. Same plusy. Minusem może być to, że trochę na pierwszy rzut oka zaciemniło nam obraz.

To teraz dla odmiany zróbmy coś co nam pozwoli uniknąć komentarzy w kodzie - komentować się będzie sam kod! Chodzi o metodę yMailer::send() i jej długi warunek, który będzie się z pewnością wydłużał (zaraz zechcesz dodać jeszcze więcej parametrów, zmienna ty, zmienna tam i w kodzie jest bałagan).

{codecitation class="brush: php;"}

```
    private function allDataSet() {
```

```
        if (false != $this->to and

            false != $this->subject and

            false != $this->content) {

            return true;

        }

    return false;

    }



    public function send() {

        $result = self::WRONG_DATA; // zakladam, ze jest cos nie tak z danymi

        if ($this->allDataSet()) {

            if (mail($this->to, $this->subject, $this->content)) {
```

```
            $result = self::SEND_SUCC;

        } else {

            $result = self::SEND_FAIL;

        }

    }

  return $result;

  }{/codecitation}
```

A co tu robią jakieś self::WRONG_DATA, self::SEND_SUCC, self::SEND_FAIL? To stałe, należy je zdefiniować na początku klasy:

{codecitation class="brush: php;"}

```
    private $content = false; // tresc wiadomosci

  const WRONG_DATA = -1;
```

```
const SEND_SUCC     = 1;

const SEND_FAIL  = 0;

public function _construct( $validation = null ) {/codecitation}
```

Po co uzywać stałych? Choćby po to, aby kod był cxzytalniejszy. Nie ma nic przyjemniejszego jak nagle zwracana wartość w stylu 214... I co to oznacza. Jeśli ją zdefiniujesz wcześniej jako

stałą czytając kod od razu wiadomo co ma się dziać. Koduj aby łatwo było czytać twój kod, bo piszesz go raz, a czytasz... na pewno więcej. Czasem setki razy.

Oczywiście, w sytuacji takiej jak tu, gdzie masz 3 zmienne do sprawdzenia to wyrzucenie tego do osobnej metody może być nadmiarowością kodu. Jednak pamiętaj, że to jest bardziej klasa przykładowa, co robić jak się za wiele rzeczy składa do kupy niż klasa, której masz używać. Choć, w sumie, jeśli by jeszcze nad nią poracować chwilę to wyszłaby bartdzo fajna klasa do obsługi maila.

Na naszej liście widnieje jeszcze 3 opcja: brak wsparcia dla wysyłania maili do więcej niż jednego odbiorcy. To już jest zadanie dla ciebie. Podpowiedź. pole $to zmień na tablicę i po prostu dopisuj nowych odbioróc do końca tablicy. A może warto jest rozszerzyć funkcjonalnośći z $to zrobić osobny obiekt? Powodzenia!

A oto cała klasa, ze wszystkimi zmianami jakich dokonaliśmy (licencja GPL):

{codecitation class="brush: php;"}<?php

```
// auto Patryk yarpo Jar, 26 IX 2009


class yMailer {


    private $validation = array( 'to' => '/@/i',


                                 'subject' => '/.{1,100}/'


                               );


    private $to         = false; // do kogo wyslac


    private $subject = false; // temat wiadomosci


    private $content = false; // tresc wiadomosci


    const WRONG_DATA = -1;


    const SEND_SUCC     = 1;


    const SEND_FAIL  = 0;
```

```
public function __construct( $validation = null ) {

    if (is_array($validation)) {

        $this->validation = $validation;

    }

}



private function validInput( $input, $name ) {

    // to pole jest ustawione w $validation => ma być sprawdzane

    if (isset($this->validation[$name])) {

        if (preg_match($this->validation[$name], $input)) {

            return true; // przeszlo test z wyrazeniem regularnym
```

```
        } else {

            return false;

        }

    } else {

        return true; // skoro nie trzeba tego sprawdzac, to zawsze jest prawidlowe

    }

}



private function add( $value, $name ) {

    if ($this->validInput($value, $name)) {

        $this->$name = $value; // przeslo test - zapiszmy wartosc

    return true; //  udało się ustawić odbiorcę.
```

```
        }

    return false; // nie udało się ustawić odbiorcy

    }




    public function recipient( $to ) {

        return $this->add($to, 'to');

    }




    public function subject( $subject ) {

        return $this->add($subject, 'subject');

    }
```

```
public function content( $content ) {

    return $this->add($content, 'content');

}




private function allDataSet() {

    if (false != $this->to and

        false != $this->subject and

        false != $this->content) {

        return true;

    }

return false;
```

```
}

public function send() {

    $result = self::WRONG_DATA; // zakladam, ze jest cos nie tak z danymi

    if ($this->allDataSet()) {

        if (mail($this->to, $this->subject, $this->content)) {

            $result = self::SEND_SUCC;

        } else {

            $result = self::SEND_FAIL;

        }

    }
```

```
    return $result;

    }

}

  $mail = new yMailer();

echo ($mail->recipient('adres(a)serwer.pl')) ? &quot;ustawiony rec<hr>&quot; : &quot;blad adresu<hr>&quot;; // zamien (a) na znak małpy

echo ($mail->subject('bbb')) ? &quot;subject jest poprawny<hr>&quot; : &quot;subject niepoprawny<hr>&quot;;

echo ($mail->content('aaa')) ? &quot;content<hr>&quot; : &quot;no content<hr>&quot;;

switch($mail->send()) {

    case yMailer::WRONG_DATA :
```

```
        echo 'Niewlasciwe dane';

    break;

    case yMailer::SEND_FAIL:

        echo&quot;Błąd&quot;;

    break;

    case yMailer::SEND_SUCC:

        echo 'sukces';

    break;

}{/codecitation}
```