EXERCISE Object Calisthenics

1) Apenas um nível de identação por método
A ideia principal é que cada método faça apenas uma coisa. Desse modo facilitamos a leitura e a manutenção do código.

Antes:

```
<?php class Board {
    public String board() {
        StringBuilder buf = new StringBuilder();

        // 0
        for (int i = 0; i < 10; i++) {
            // 1
            for (int j = 0; j < 10; j++) {
                // 2
                buf.append(data[i][j]);
            }
            buf.append("\n");
        }

        return buf.toString();
    }
}
?>
```

Depois:

```
<?php class Board {
    public String board() {
        StringBuilder buf = new StringBuilder();

        collectRows(buf);

        return buf.toString();
    }

    private void collectRows(StringBuilder buf) {
        for (int i = 0; i < 10; i++) {
            collectRow(buf, i);
        }
    }

    private void collectRow(StringBuilder buf, int row) {
        for (int i = 0; i < 10; i++) {
            buf.append(data[row][i]);
        }

        buf.append("\n");
```

```
    }
}
?>
```

2) Não use else
A ideia principal é negar tudo que for possível no método antes. Dessa forma assumimos retornos antecipados e definimos um fluxo de trabalho padrão.

Antes:
```
<?php
public void login(String username, String password) {
    if (userRepository.isValid(username, password)) {
        redirect("homepage");
    } else {
        addFlash("error", "Bad credentials");

        redirect("login");
    }
}
?>
```

Depois:
```
<?php
public void login(String username, String password) {
    if (userRepository.isValid(username, password)) {
        return redirect("homepage");
    }

    addFlash("error", "Bad credentials");

    return redirect("login");
}
?>
```

3) Envolver os tipos primitivos
A ideia principal é encapsular todos os tipos primitivos dentro dos objetos.

Antes:
```
<?php
class Order
{
    public function notifyBuyer($email)
    {
        if (filter_var($email, FILTER_VALIDATE_EMAIL) === false) {
            throw new InvalidEmailException;
        }
```

```
        return $this->repository->sendEmail($email);
    }
}
?>
```

Depois:

```
<?php
class Email {
    public $email;

    public function __construct($email)
    {
        if (filter_var($email, FILTER_VALIDATE_EMAIL) === false) {
            throw new InvalidEmailException;
        }

        return $this->email = $email;
    }
}

class Order
{
    public function notifyBuyer($email)
    {
        return $this->repository->sendEmail(new Email($email));
    }
}
?>
```

4) Envolver os collections em classes

A ideia inicial dessa regra diz que se você tiver um conjunto de elementos e quiser manipulá-los, é necessário criar uma classe dedicada (collection) apenas para este conjunto. Assim, ao atualizar aquele valor, com certeza será em sua collection.

Seguindo o comportamento dessa regra, você deixa os comportamentos relacionados.

Antes:

```
<?
function Company(name) {

    this.companyName = name;

    var developerTeam = new Array()
    var salesTeam = new Array()

    this.addDeveloper = function(employee) {
```

```
        if (!(developerTeam.includes(employee) || developerTeam.length ==
5)) {
            developerTeam.push(employee)
        }
    }

    this.addSeller = function(employee) {
        if (!(salesTeam.includes(employee) || salesTeam.length == 5)) {
            salesTeam.push(employee)
        }
    }

    this.toString = function() {
        return salesTeam + developerTeam;
    }
}
?>

Depois:
<?
function Company(name) {

    this.companyName = name;

    var developerTeam = new Team("Developer Division")

    var salesTeam = new Team("Sales Department")

    this.addDeveloper = function(employee) {
        developerTeam.addEmployee(employee)
    }

    this.addSeller = function(employee) {
        salesTeam.addEmployee(employee)
    }

}

function Team(name) {

    this.teamName = name
    var employees = new Array()
    const TEAM_SIZE = 5

    this.addEmployee = function(employee) {

        if (employees.includes(employee)) {
            throw new Error("Employee " + employee + "already present.")
```

```
        }

        if (employees.length == TEAM_SIZE) {
            throw new Error("Team full, max " + TEAM_SIZE + " employees.")
        }

        employees.push(employee)

    }

}
?>
```

5) Usar apenas um "ponto" por linha
Esse “ponto” é o que usamos para chamar métodos em Java ou C#; no caso do PHP seria uma seta.

Antes:

```
<?
package chess

type piece struct {
    representation string
}

type location struct {
  current *piece
}

type board struct {
    locations []*location
}

func NewLocation(piece *piece) *location {
  return &location{current: piece}
}

func NewPiece(representation string) *piece {
    return &piece{representation: representation}
}

func NewBoard() *board {
    locations := []*location{
        NewLocation(NewPiece("London")),
        NewLocation(NewPiece("New York")),
        NewLocation(NewPiece("Dubai")),
    }
```

```
    return &board{
        locations: locations,
    }
}

func (b *board) squares() []*location {
    return b.locations
}

func (b *board) BoardRepresentation() string {
    var buffer = &bytes.Buffer{}
    for _, l := range b.squares() {
        buffer.WriteString(l.current.representation[0:1])
    }
    return buffer.String()
}
?>
```

Depois:

```
<?
package chess

import "bytes"

type piece struct {
    representation string
}

type location struct {
    current *piece
}

type board struct {
    locations []*location
}

func NewPiece(representation string) *piece {
    return &piece{representation: representation}
}

func (p *piece) character() string {
    return p.representation[0:1]
}

func (p *piece) addTo(buffer *bytes.Buffer) {
    buffer.WriteString(p.character())
}
```

```
func NewLocation(piece *piece) *location {
   return &location{current: piece}
}

func (l *location) addTo(buffer *bytes.Buffer) {
   l.current.addTo(buffer)
}

func NewBoard() *board {
   locations := []*location{
      NewLocation(NewPiece("London")),
      NewLocation(NewPiece("New York")),
      NewLocation(NewPiece("Dubai")),
   }
   return &board{
      locations: locations,
   }
}

func (b *board) squares() []*location {
   return b.locations
}

func (b *board) BoardRepresentation() string {
   var buffer = &bytes.Buffer{}
   for _, l := range b.squares() {
      l.addTo(buffer)
   }
   return buffer.String()
}
?>
```

6) Não abreviar
Essa regra visa melhor entendimento por parte de quem está visualizando o código.

7) Manter todas as classes pequenas
É recomendável que uma classe tenha no máximo 50 linhas, e que os pacotes não tenham mais do que 10 arquivos.
Geralmente,quando criamos uma classe com mais de 50 linhas, atribuímos à ela mais responsabilidades, tornando-as mais difíceis de se entender e também de se reutilizar.

Classes e pacotes devem ser coesos e ter um propósito, e esse propósito deve ser fácil de se entender.

Antes:
<?

```
type Repository interface {
  Find(id entity.ID) (*entity.User, error)
  FindByEmail(email string) (*entity.User, error)
  FindByChangePasswordHash(hash string) (*entity.User, error)
  FindByValidationHash(hash string) (*entity.User, error)
  FindByChallengeSubmissionHash(hash string) (*entity.User, error)
  FindByNickname(nickname string) (*entity.User, error)
  FindAll() ([]*entity.User, error)
  Update(user *entity.User) error
  Store(user *entity.User) (entity.ID, error)
  Remove(id entity.ID) error
}
?>
```

Depois

```
<?
type Reader interface {
  Find(id entity.ID) (*entity.User, error)
  FindByEmail(email string) (*entity.User, error)
  FindByChangePasswordHash(hash string) (*entity.User, error)
  FindByValidationHash(hash string) (*entity.User, error)
  FindByChallengeSubmissionHash(hash string) (*entity.User, error)
  FindByNickname(nickname string) (*entity.User, error)
  FindAll() ([]*entity.User, error)
}

type Writer interface {
  Update(user *entity.User) error
  Store(user *entity.User) (entity.ID, error)
  Remove(id entity.ID) error
}

type Repository interface {
  Reader
  Writer
}
?>
```

8) Não ter mais que duas variáveis de instância na classe

Antes:

```
<?
class Name {
  constructor(first, middle, last) {
    this._first = first;
    this._middle = middle;
    this._last = last;
  }
```

```
}
?>
```

Depois:

```
<?
class Name {
  constructor(givenName, surname) {
    this._givenName = givenName;
    this._surname = surname;
  }
}

class GivenName {
  constructor(...names) {
    this._names = names;
  }
}

class Surname {
  constructor(familyName) {
    this._familyName = familyName;
  }
}
?>
```

9) Não usar Getters ou Setters

Os getters e setters são removidos para poder adicionar decisões no próprio objeto.

Antes

```
<?php
class Buyer {
    protected $name;

    protected $purchases;

    public function getName() {/**/}
    public function setName($name) {/**/}

    public function getPurchases() {/**/}
    public function setPurchases($purchases) {/**/}
}
?>
```

Depois:

```
<?php
class Buyer {
    protected $name;
```

```
    protected $purchases;

    public function addNewPurchase()
    {
        $this->purchases++;
    }
}
?>
```