# OBJETS CALISTHENICS

ESTEVES PEDRO – 03.03.2022

# QUOI ? POURQUOI ?

- Jeff Bay - *The ThoughtWorks Anthology – exercices*
- 9 règles de codages à suivre pendant l'écriture du code
- Orienté sur 4 axes
  - Testabilité
  - Maintenabilité
  - Lisibilité
  - Compréhensibilité
- Améliorer la qualité du code

# 9 RÈGLES

- 1 niveau d'indentation par méthode
- Ne pas utiliser les "Else"
- Encapsuler les primitives et les strings dans des objets
- Collections de première classe
- Un seul point par ligne
- Bannir les abréviations
- Classes de petites tailles
- 2 variables d'instance maximum
- Eviter les getter / setter

# 1 NIVEAU D'INDENTATION

```php
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_pasaword_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

## Techniques

Return

Extraction de méthode

## Avantages

Méthodes concentrées sur une seule chose

# NE PAS UTILISER LES "ELSE"

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

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

**Devient :**

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

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

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

**Techniques**

Return

Polymorphisme

**Avantages**

Une seule voie d'exécution principale

# ENCAPSULER LES PRIMITIVES ET STRINGS DANS DES OBJETS

```
public class Address : ValueObject
{
    public String Street { get; private set; }
    public String City { get; private set; }
    public String State { get; private set; }
    public String Country { get; private set; }
    public String ZipCode { get; private set; }

    public Address() { }

    public Address(string street, string city, string state, string country
    {
        Street = street;
        City = city;
        State = state;
        Country = country;
        ZipCode = zipcode;
    }
```

**Techniques**

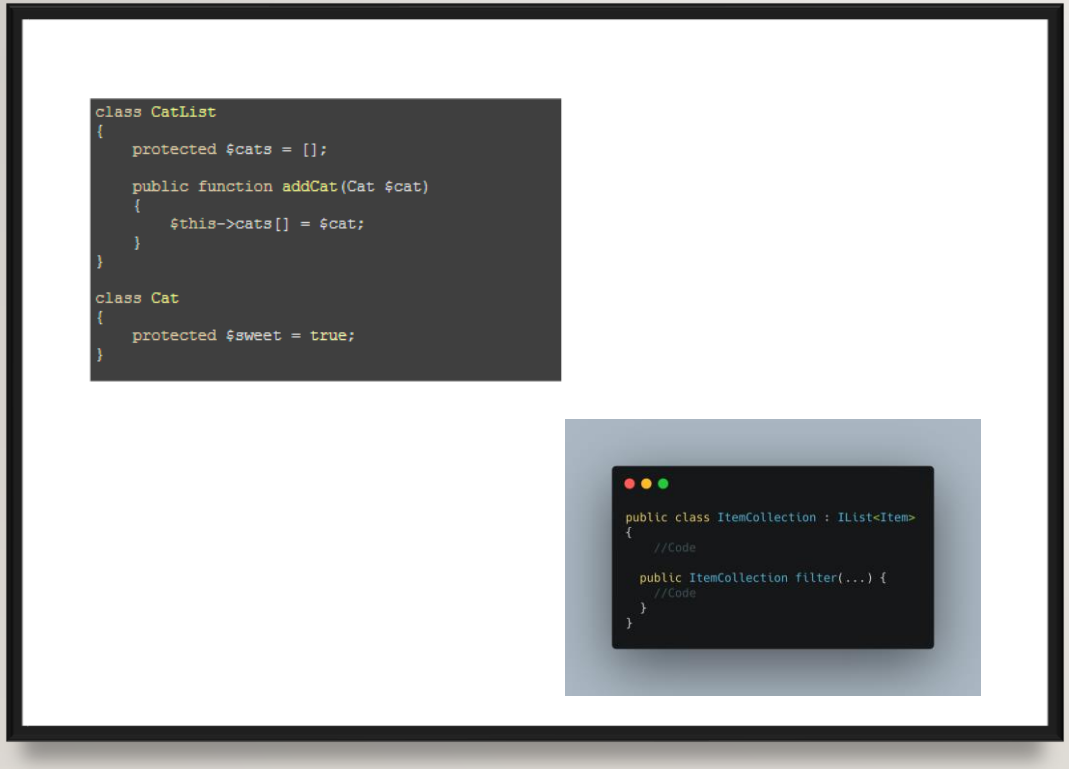
Value object

**Avantages**

Code plus explicite

Intelligence portée par l'objet (règles)

# COLLECTIONS DE PREMIÈRE CLASSES



```php
class CatList
{
    protected $cats = [];

    public function addCat(Cat $cat)
    {
        $this->cats[] = $cat;
    }
}

class Cat
{
    protected $sweet = true;
}
```

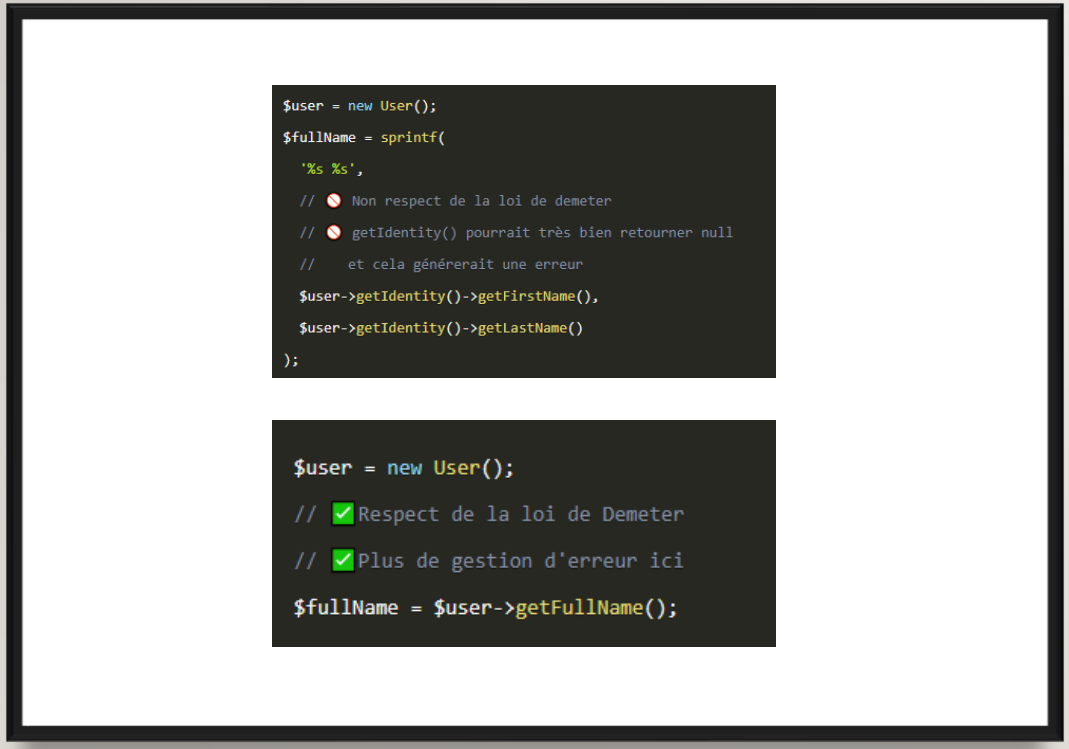
```php
public class ItemCollection : IList<Item>
{
    //Code

  public ItemCollection filter(...) {
    //Code
  }
}
```

- Classe qui contient une collection ne doit avoir aucun attribut

- Comportement spécifique à la collection à un seul endroit

(recherche, tri, règle d'insertion, …)

# UN SEUL POINT PAR LIGNE



```php
$user = new User();
$fullName = sprintf(
  '%s %s',
  // 🚫 Non respect de la loi de demeter
  // 🚫 getIdentity() pourrait très bien retourner null
  //    et cela générerait une erreur
  $user->getIdentity()->getFirstName(),
  $user->getIdentity()->getLastName()
);
```

```php
$user = new User();
// ✅Respect de la loi de Demeter
// ✅Plus de gestion d'erreur ici
$fullName = $user->getFullName();
```

- Loi de Demeter "ne parler qu'a ses amis immediats"
- Demande un comportement, pas la representation interne (Chien qui remue la queue)

**Avantages**

- Test unitaires : moins de dépendances

# BANNIR LES ABREUVIATIONS

```
var factu = compta.GenFactu(42);
```

- Complique la lisibilité
- Trop long sous-entends un cas trop complexe

**Avantages**

- Meilleure compréhension

# CLASSES DE PETITES TAILLES

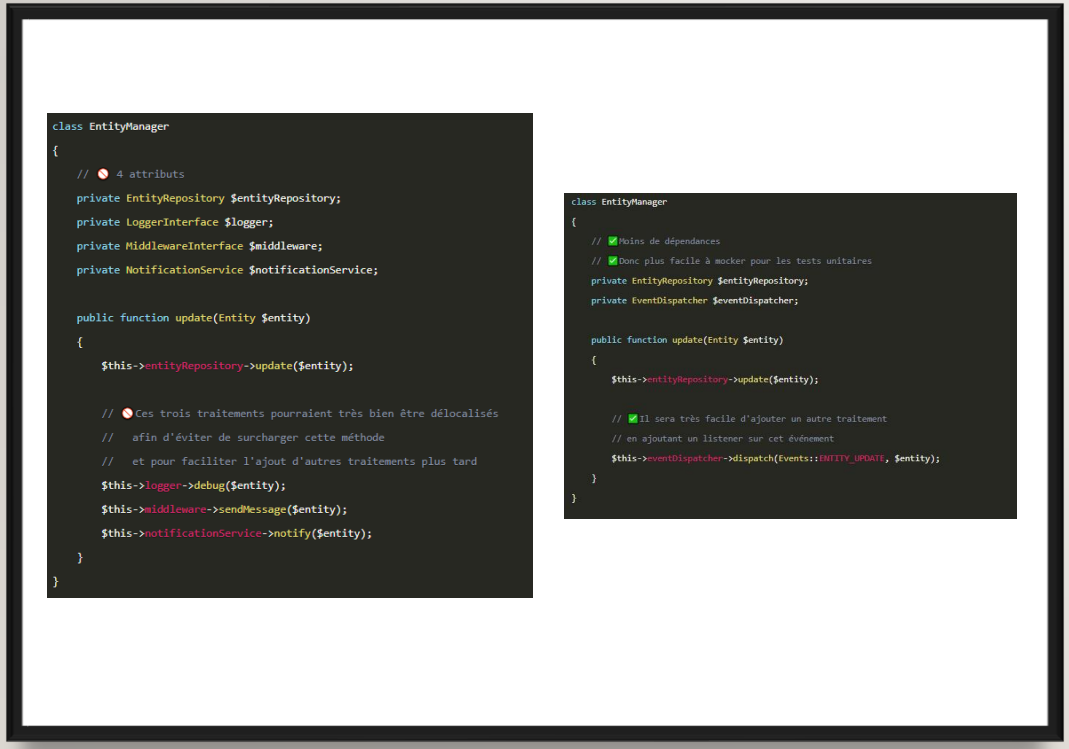
**Recommendation**

- 10 classes par namespace
- 10 méthodes par classe
- 2 arguments par méthode
- 50 lignes par classes

**Avantages**

- Permet d'identifier les responsabilités
- Plus simple à maintenir

# 2 VARIABLES D'INSTANCES MAXIMUM



```php
class EntityManager
{
    // 🚫 4 attributs
    private EntityRepository $entityRepository;
    private LoggerInterface $logger;
    private MiddlewareInterface $middleware;
    private NotificationService $notificationService;

    public function update(Entity $entity)
    {
        $this->entityRepository->update($entity);

        // 🚫 Ces trois traitements pourraient très bien être délocalisés
        //   afin d'éviter de surcharger cette méthode
        //   et pour faciliter l'ajout d'autres traitements plus tard
        $this->logger->debug($entity);
        $this->middleware->sendMessage($entity);
        $this->notificationService->notify($entity);
    }
}
```

```php
class EntityManager
{
    // ✅ Moins de dépendances
    // ✅ Donc plus facile à mocker pour les tests unitaires
    private EntityRepository $entityRepository;
    private EventDispatcher $eventDispatcher;

    public function update(Entity $entity)
    {
        $this->entityRepository->update($entity);

        // ✅ Il sera très facile d'ajouter un autre traitement
        // en ajoutant un listener sur cet événement
        $this->eventDispatcher->dispatch(Events::ENTITY_UPDATE, $entity);
    }
}
```

- Séparation des responsabilités

**Avantages**

- Limitation des dépendances
- Tests unitaires

# EVITER LES GETTERS / SETTERS

```php
public function diceRoll(int $score): void
{
    $actualScore = $this->score->getScore();
    // 🚫 On modifie en dehors de l'objet sa valeur pour ensuite lui "forcer" le résultat
    $newScore = $actualScore + $score;
    $this->score->setScore($newScore);
}
```

```php
class Game
{
    private Score $score;

    public function diceRoll(Score $score): void
    {
        $this->score->addScore($score);
    }
}
```

- "Tell, don't ask"
- Placer l'intelligence au sein des objets

**Avantages**

- Meilleures lisibilité
- Responsabilité au sein des objets

# FEEDBACK KATA

Importance d'y penser en amont car même avec petit projet, peut nécessité de gros projets de refactorisation (encapsulation des primitifs / responsabilité des objets)

Un simple refacto pour eviter un if else… peut avoir un impact important