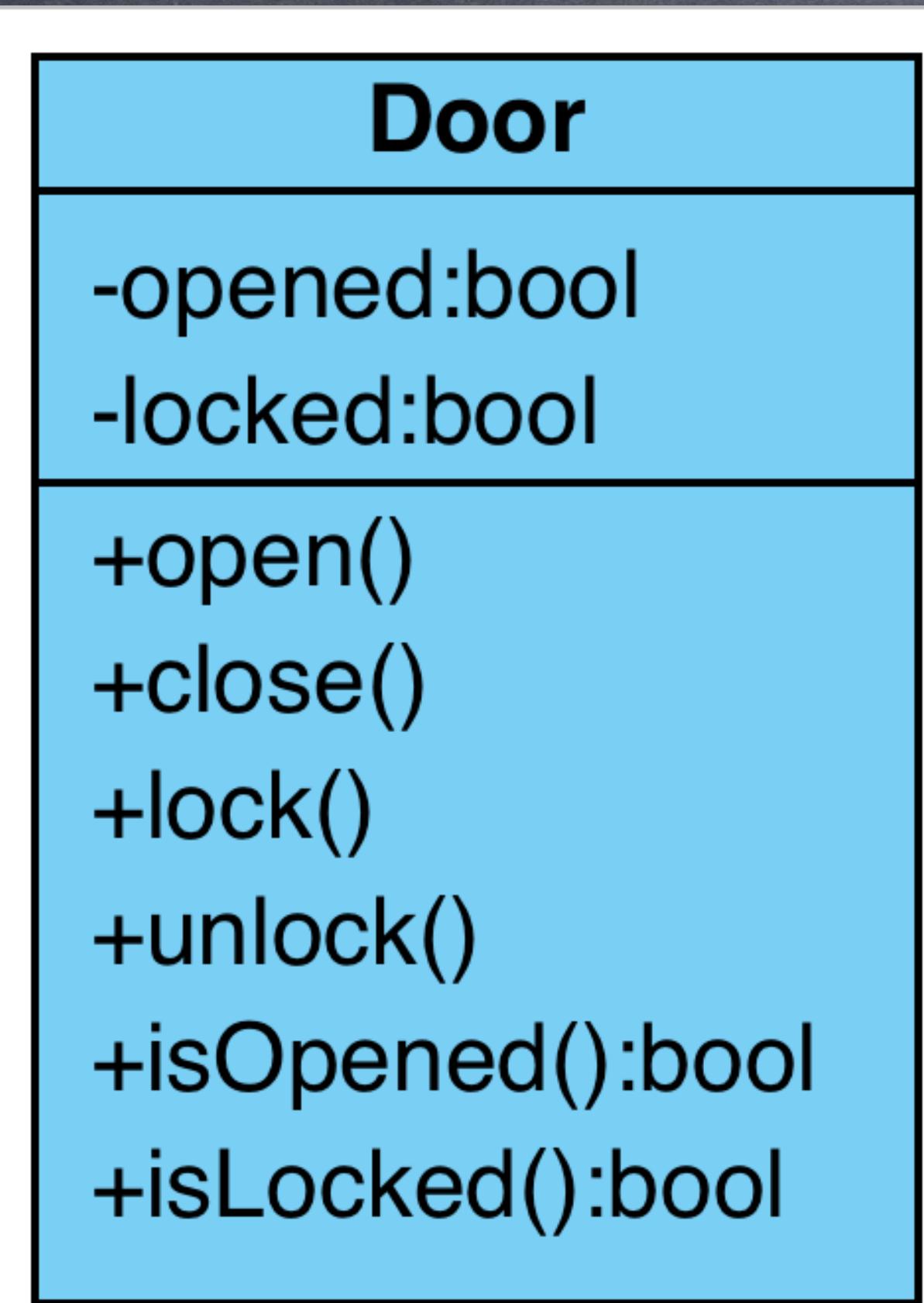


PHP Objet

Structure de données possédant :

- ✓ Des propriétés (les données)
- ✓ Un état (valeurs des propriétés)
- ✓ Des méthodes (pour interagir avec les propriétés)



Etats :

Ouvert/fermé

verrouillé/déverrouillé

Les méthodes se chargent de modifier l'état et assurent que celui-ci reste cohérent (pas de porte ouverte et verrouillée en même temps)

Un modèle, un moule qui permet de créer (instancier) des objets

Une classe combine les données et les comportements (méthodes)
qui manipulent ces données

```
<?php
class Color
{
    private int $red;
    private int $green;
    private int $blue;

    public function __construct(int $red, int $green, int $blue)
    {
        $this->red = $red;
        $this->green = $green;
        $this->blue = $blue;
    }
}
```

classes/Color.php

```
public function getRGB(): string {
    return "rgb($this->red, $this->green, $this->blue)";
}

private function toHex(int $num): string {
    return str_pad(dechex($num), 2, '0', STR_PAD_LEFT);
}
public function getHex(): string {
    return "#{$this->toHex($this->red)}{$this->toHex($this->green)}{$this->toHex($this->blue)}";
}
```

```
<?php
include 'classes/Color.php';

$color = new Color(255, 255, 0);

echo $color->getRGB();

echo $color->getHex();
```

test-color.php

Door

-opened:bool
-locked:bool

+open()
+close()
+lock()
+unlock()
+isOpened():bool
+isLocked():bool

Ecrire le code de la classe Door

```
class Door
{
    private bool $opened = false;
    private bool $locked = false;

    public function open(): void
    {
        if(! $this->opened && ! $this->locked) {
            $this->opened = true;
        }
    }

    public function close(): void
    {
        if($this->opened) {
            $this->opened = false;
        }
    }

    public function lock(): void
    {
        if(! $this->opened ) {
            $this->locked = true;
        }
    }

    public function unlock(): void
    {
        if(! $this->opened) {
            $this->locked = false;
        }
    }

    public function isLocked(): bool
    {
        return $this->locked;
    }

    public function isOpen(): bool
    {
        return $this->opened;
    }
}
```

```
<?php
class Color
{
    public function __construct
    (
        private int $red,
        private int $green,
        private int $blue
    )
    {}
}
```

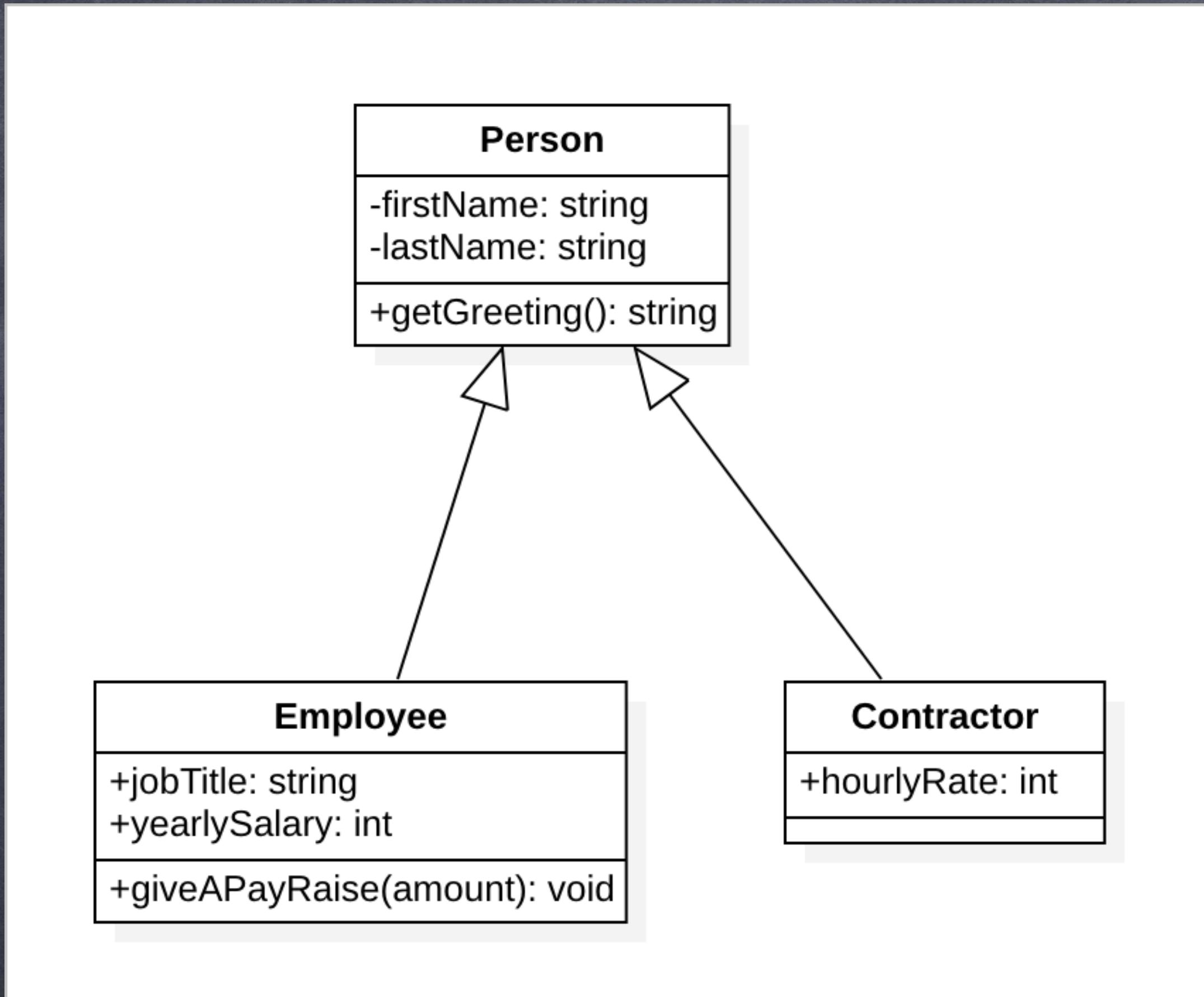
Les propriétés sont déclarées dans le constructeur

- ✓ Le constructeur initialise l'objet
- ✓ Les propriétés représentent l'état de l'objet
- ✓ Les méthodes manipulent les propriétés

- ✓ Encapsulation
- ✓ Héritage
- ✓ Polymorphisme

- ✓ Restreindre l'accès aux propriétés et méthodes
- ✓ Protéger l'intégrité des données
- ✓ Définir une interface publique et masquer la mécanique interne
(exemple: le poste radio, mêmes contrôles pour une radio analogique et une radio numérique)

- ✓ public (accessible à toute le monde)
- ✓ protected (accessible uniquement à la classe)
- ✓ private (accessible à la classe et à ses enfants)



Les employés et les indépendants sont tous des personnes. Ils partagent des attributs (nom et prénoms) et des méthodes mais possèdent des caractéristiques propres

```
class Person
{
    private string $firstName;
    private string $lastName;

    public function __construct(string $firstName, string $lastName)
    {
        $this->firstName = $firstName;
        $this->lastName = $lastName;
    }

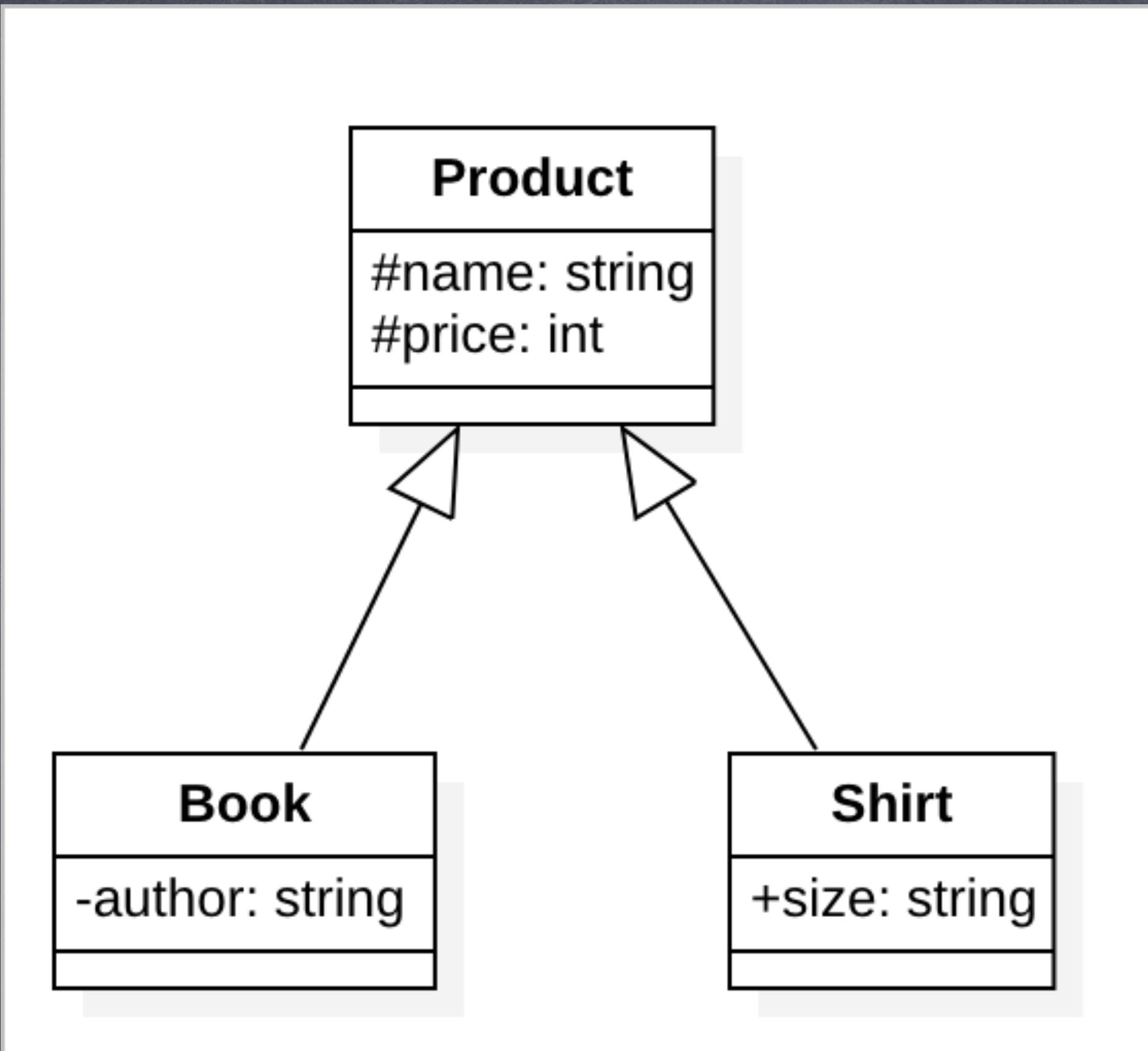
    public function getGreeting(): string{
        return "Bonjour mon nom est {$this->firstName} {$this->lastName}";
    }
}
```

classes/Person.php

```
class Employee extends Person {  
    private string $jobTitle;  
    private int $yearlySalary;  
  
    public function __construct(  
        string $firstName,  
        string $lastName,  
        string $jobTitle,  
        int $yearlySalary  
    ) {  
        parent::__construct($firstName, $lastName);  
        $this->jobTitle = $jobTitle;  
        $this->yearlySalary = $yearlySalary;  
    }  
  
    public function getGreeting(): string{  
        return parent::getGreeting().  
            " je suis employé en tant que $this->jobTitle  
            et je gagne $this->yearlySalary";  
    }  
  
    public function giveAPayRaise(int $amount): int{  
        return $this->yearlySalary += $amount;  
    }  
}
```

```
class Contractor extends Person {  
    private int $hourlyRate;  
  
    public function __construct(  
        string $firstName,  
        string $lastName,  
        int $hourlyRate  
    ) {  
        parent::__construct($firstName, $lastName);  
        $this->hourlyRate = $hourlyRate;  
    }  
  
    public function getGreeting(): string{  
        return parent::getGreeting().  
            " je suis indépendant  
            et je facture $this->hourlyRate de l'heure";  
    }  
}
```

Créer une classe Student qui hérite de Person et possède une propriété "schoolName"



La chemise et le livre sont des produits. La visibilité `protected` permet d'accéder directement aux propriétés du parent

```
class Product
{
    protected string $name;
    protected int $price;

    public function __construct(string $name, int $price)
    {
        $this->name = $name;
        $this->price = $price;
    }
}

class Book extends Product {
    private string $author;

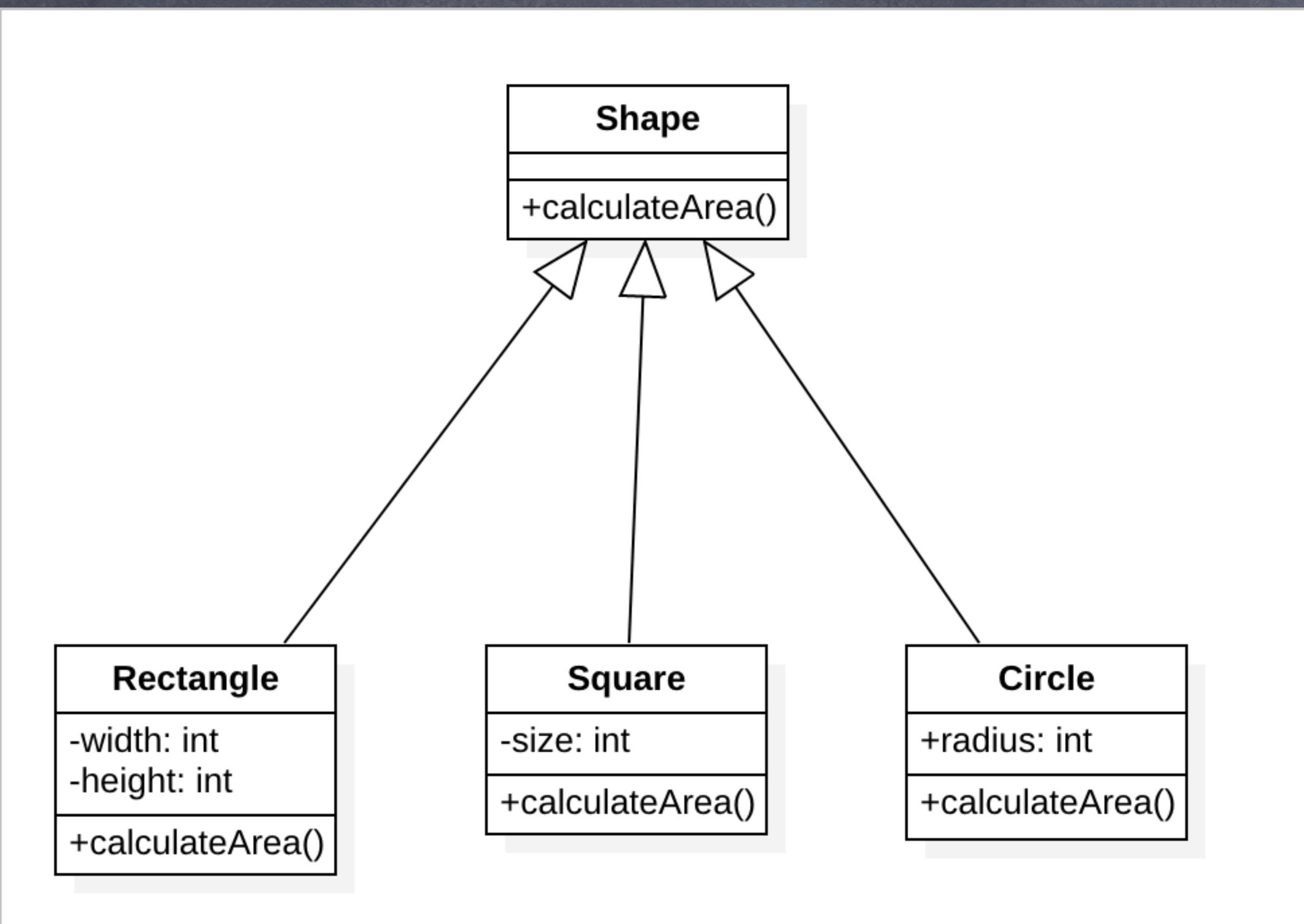
    public function __construct(
        string $name, string $author, int $price)
    {
        parent::__construct($name, $price);
        $this->author = $author;
    }

    public function getDetails(): string{
        return "un livre intitulé $this->name
            par $this->author coutant $this->price";
    }
}

class Shirt extends Product {
    private string $size;

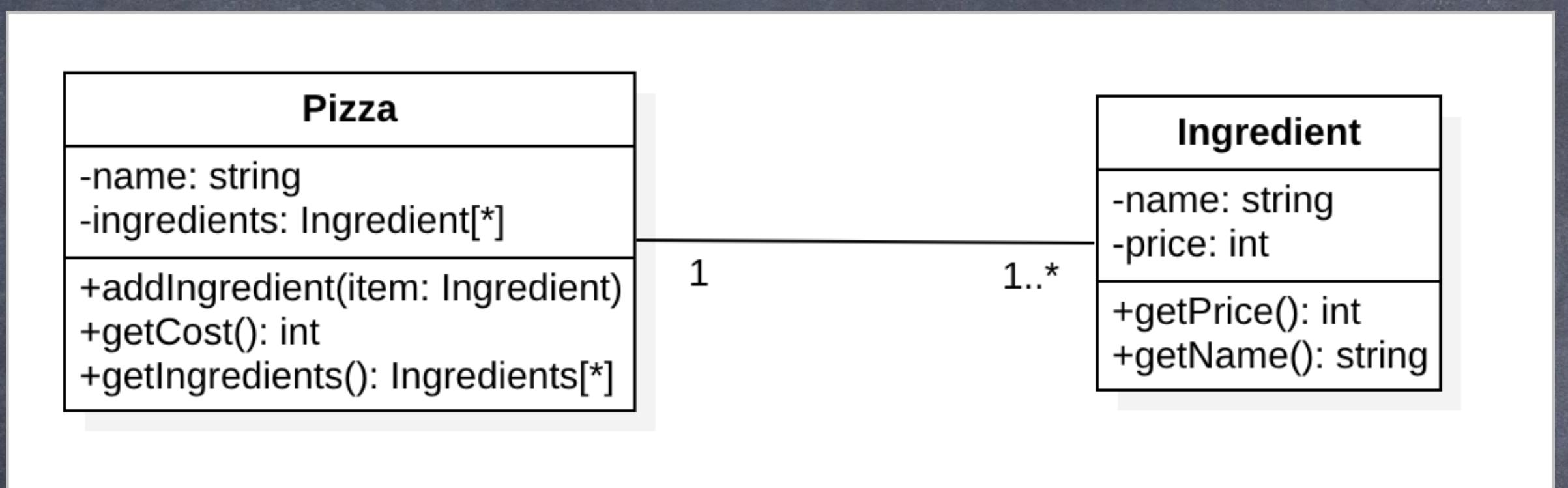
    public function __construct(
        string $name, string $size, int $price)
    {
        parent::__construct($name, $price);
        $this->size = $size;
    }

    public function getDetails(): string{
        return "une chemise taille $this->size
            modèle $this->name coutant $this->price";
    }
}
```



Le rectangle, le carré et le cercle sont des formes géométriques, elles partagent une méthode pour calculer l'aire mais l'implémentation est différente

Ecrire le code de ces classes



```
class Pizza
{
    private Array $ingredients = [];

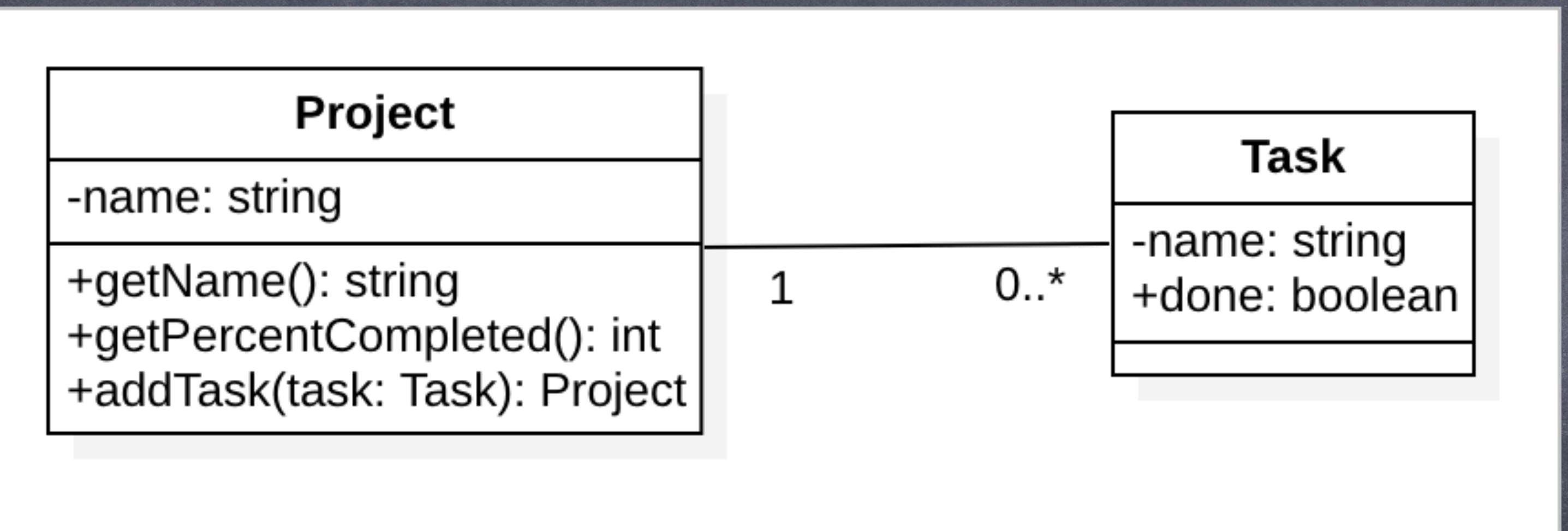
    public function __construct(private string $name){}
    public function addIngredient(Ingredient $ingredient)
    {
        $this->ingredients[] = $ingredient;
    }
    public function getIngredients(): Array {
        return $this->ingredients;
    }

    public function getCost(): int {
        $ingredientPrices = array_map(
            static function ($item){
                return $item->getPrice();
            },
            $this->ingredients);

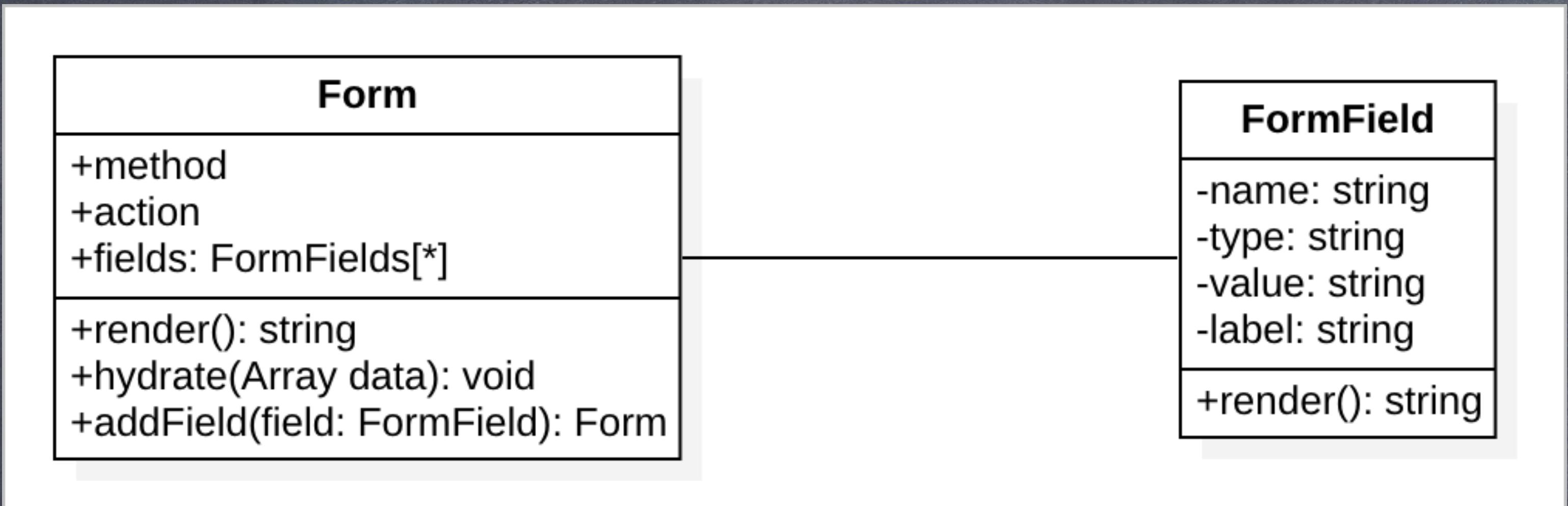
        return array_sum($ingredientPrices);
    }
}
```

```
class Ingredient {
    public function __construct(
        private string $name,
        private int $price){}
    public function getPrice(): int
    {
        return $this->price;
    }
}
```

Ecrire le code de ces classes



Ecrire le code de ces classes



Utiliser la classe Form pour afficher et traiter un formulaire

- ✓ Une propriété ou une méthode d'une classe qui n'a pas besoin d'instance
- ✓ Dans le cas d'une propriété, la valeur est donc la même pour toutes les instances
- ✓ Ne pas abuser des méthodes statiques

```
class DBConnection
{
    private static PDO $connection;

    public static function getConnection(): PDO {
        if (!self::$connection) {
            self::$connection = new PDO('mysql:host=localhost;dbname=mydb', 'username', 'password');
        }
        return self::$connection;
    }
}
```

Cette classe utilise le pattern Singleton pour toujours retourner la même instance de PDO

```
include 'classes/DBConnection.php';

DBConnection::getConnection();
```

- ✓ Quand on veut utiliser un classe comme un conteneur de méthodes (exemple : l'objet Math de Javascript)
- ✓ Quand on veut stocker des informations qui doivent être disponibles partout dans l'application (configuration, cache, état de l'application)
- ✓ Quand l'instanciation n'a aucun intérêt (pattern factory)

- ✓ Couplage fort, forte dépendance des classes utilisatrices
- ✓ Peu de flexibilité, incompatible avec le principe open/close
- ✓ Les dépendances ne sont pas évidentes (il faut lire tout le code pour le repérer)
- ✓ Difficile de tester une classe en isolation si elle dépend de méthodes statiques
- ✓ Les propriétés statiques sont globales, elles peuvent provoquer des effets de bord
- ✓ On ne fait plus de l'orienté objet, mais du procédural déguisé en objet

- ✓ Ressemble aux propriétés statiques
- ✓ Ne peut être modifiée (c'est une constante)

```
class Circle {  
    const float PI = 3.14159;  
  
    public function calculateArea(float $radius): float {  
        return self::PI * $radius**2;  
    }  
}
```

- ✓ Un contrat qui détermine des méthodes et/ou des propriétés qu'une classe doit implémenter
- ✓ La classe doit respecter la signature des méthodes de ses interfaces
- ✓ Une classe peut implémenter plusieurs interfaces
- ✓ On peut (doit) typer sur une interface

```
class PaypalMethod {
    public function pay(int $amount)
    {
        echo "Paiement par Paypal";
    }
}

class CreditCardMethod {
    public function pay(int $amount)
    {
        echo "Paiement par carte de crédit";
    }
}

class BuyProduct
{
    public function pay($paymentMethod, int $total)
    {
        $paymentMethod->pay($total);
    }
}
```

Impossible de typer le mode de paiement

```
interface PaymentMethodInterface {
    public function pay(int $amount);
}

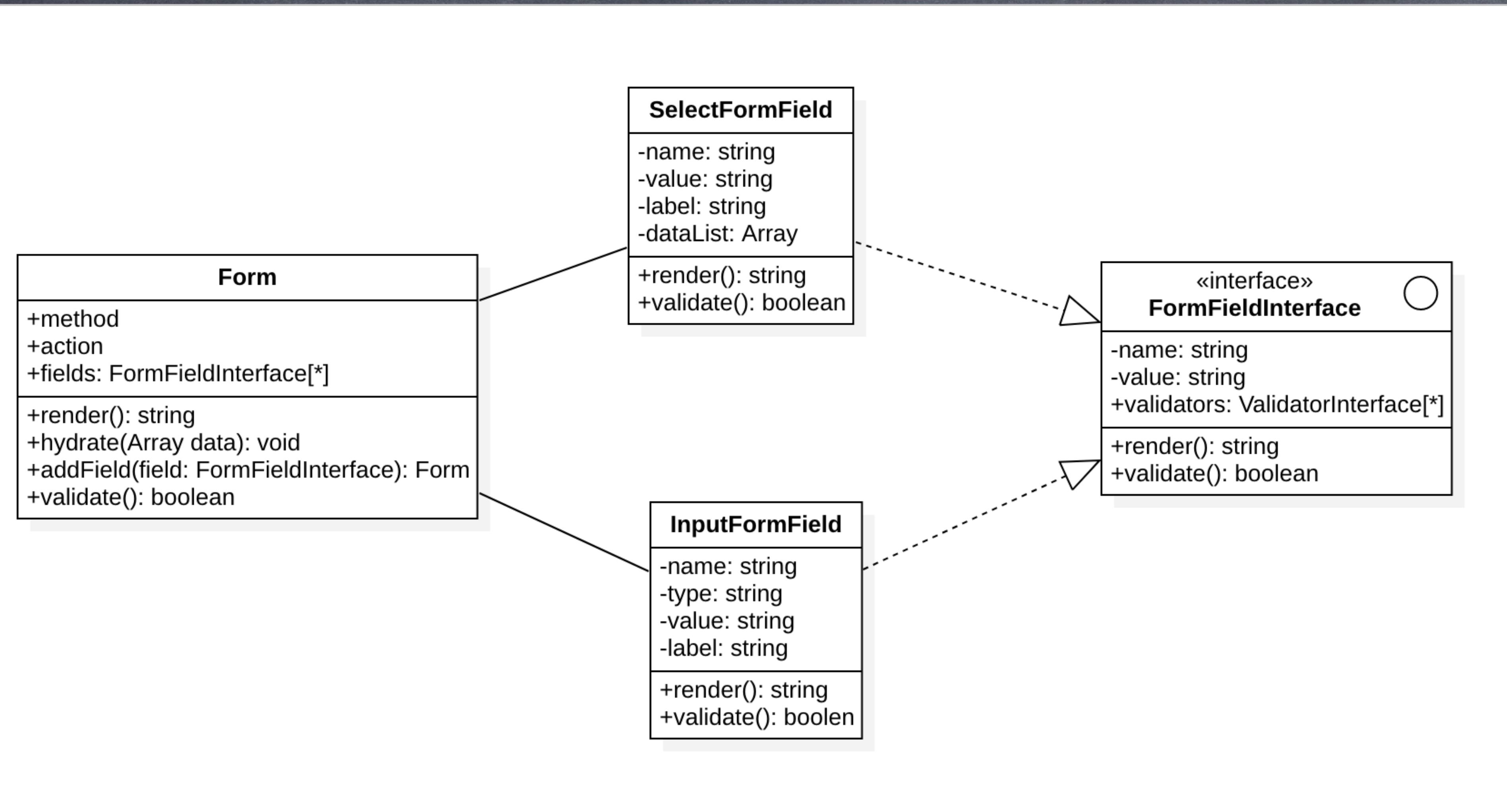
class PaypalMethod implements PaymentMethodInterface {
    public function pay(int $amount)
    {
        echo "Paiement par Paypal";
    }
}

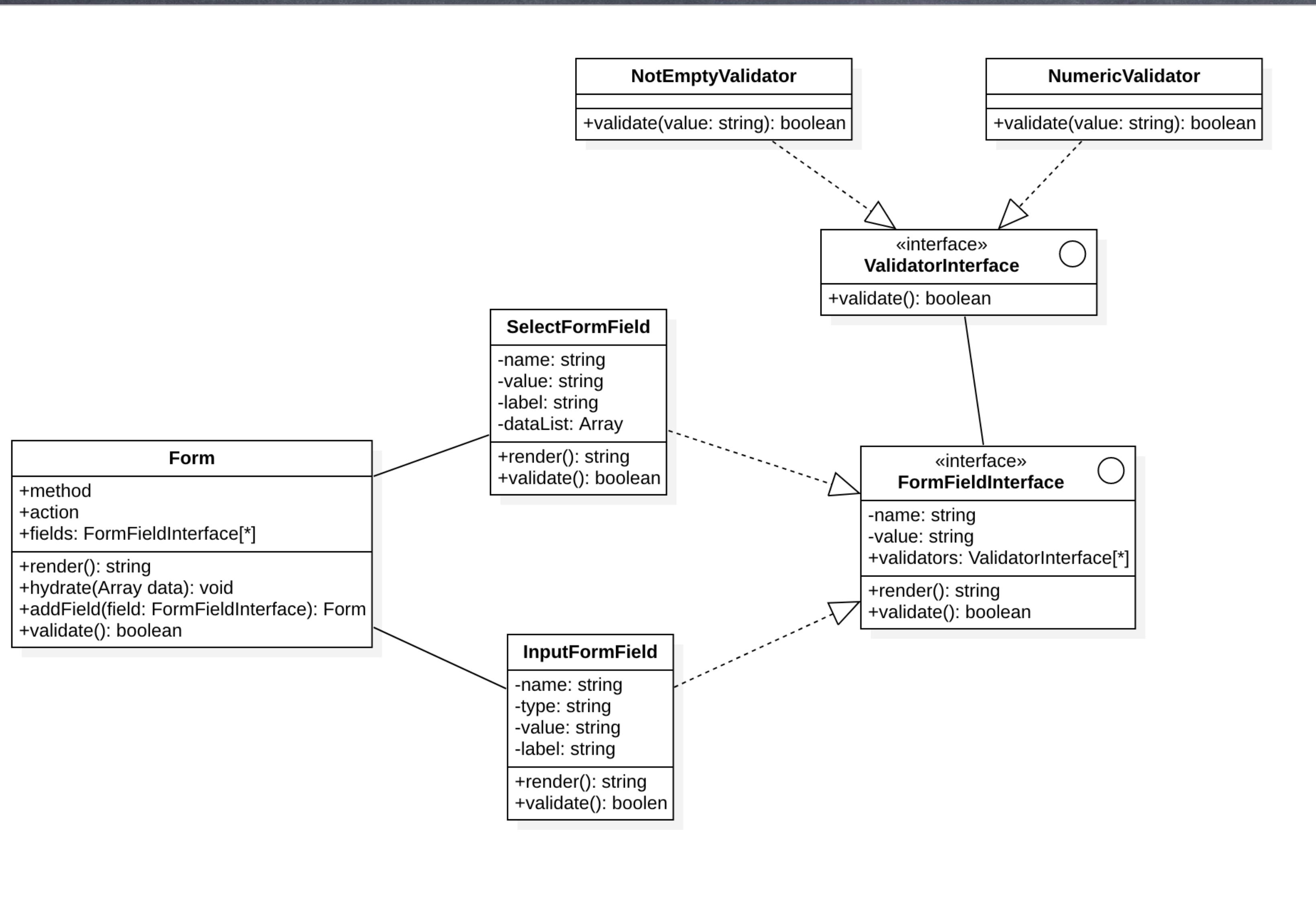
class CreditCardMethod implements PaymentMethodInterface {
    public function pay(int $amount)
    {
        echo "Paiement par carte de crédit";
    }
}

class BuyProduct
{
    public function pay(
        PaymentMethodInterface $paymentMethod, int $total)
    {
        $paymentMethod->pay($total);
    }
}
```

Le type est sur l'interface
et non une classe concrète

Exercice : Amélioration de la classe Form

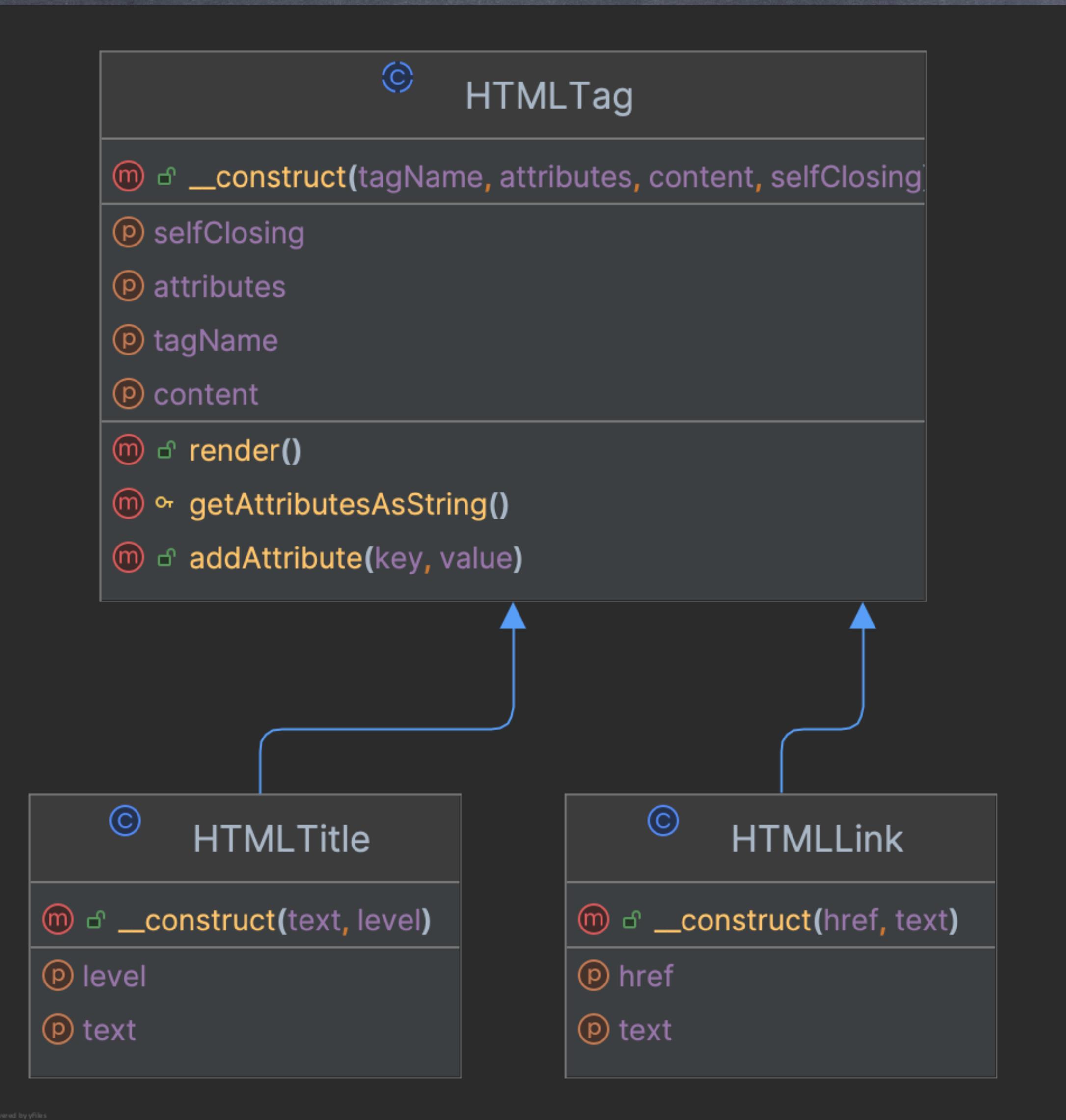




- ✓ On peut ajouter un nouveau type de champ sans avoir à modifier le code existant
- ✓ On peut ajouter une nouvelle règle de validation
- ✓ Tant que les nouvelles classes respectent les interfaces requises elles sont utilisables

- ✓ Similaire aux interfaces mais peuvent contenir des implémentations
- ✓ Les classes abstraites ne peuvent être instanciées

Un exemple



La classe abstraite

```
abstract class HTMLTag
{
    public function __construct(
        private string $tagName,
        private array $attributes = [],
        private string $content = "",
        private bool $selfClosing = false
    )
    {}

    protected function getAttributesAsString(){
        return implode(" ", array_map(
            static function ($key, $value){
                return "$key=\"$value\"";
            },
            $this->attributes
        ));
    }

    public function render(): string
    {
        $attributes = $this->getAttributesAsString();

        if($this->selfClosing) {
            return "<$this->tagName $attributes />";
        }

        return "<$this->tagName $attributes>$this->content</$this->tagName>";;
    }

    public function addAttribute(string $key, string $value)
    {
        $this->attributes[$key] = $value;
    }
}
```

Cette classe fournit une
implémentation pour gérer des
balises HTML

```
class HTMLTitle extends HTMLTag
{
    public function __construct(
        private string $text,
        private int $level = 1
    )
    {
        parent::__construct("h".$this->level, [], $this->text);
    }
}

class HTMLLink extends HTMLTag
{
    public function __construct(
        private string $href,
        private string $text
    )
    {
        parent::__construct("a", ["href" => $this->href], $this->text);
    }
}
```

En passant les propriétés
de la classe abstraite
en visibilité protected
peut-on simplifier ce code ?

- ✓ La classe abstraite permet de partager du code générique
- ✓ Elle peut se substituer à une interface
- ✓ Les classes concrètes spécialisent la classe abstraite et peuvent parfois simplifier son interface publique

- ✓ Partage de code entre des classes sans passer par l'héritage
- ✓ Spécificité de PHP
- ✓ Peut contenir des propriétés et des méthodes

Un exemple

```
trait Loggable {
    public function log(string $message): void {
        $timestamp = date('Y-m-d H:i:s');
        file_put_contents('app.log', "[{$timestamp}] {$message}\n", FILE_APPEND);
    }
}

class User {
    use Loggable;

    public function register(): void {
        $this->log("Nouvel inscrit");
    }
}

class Product {
    use Loggable;

    public function add(): void {
        $this->log("Nouveau produit ajouté");
    }
}
```

```
trait HasRoles {
    protected $roles = [];
    public function assignRole(string $role): void {
        // ici le code
    }

    public function hasRole(string $role): bool {
        // ici le code
    }

    public function getroles(): array {
        // ici le code
    }
}

trait HasPermissions {
    protected array $permissions = [];
    public function givePermissionTo(string $permission): void {
        // ici le code
    }

    public function hasPermission(string $permission): bool {
        // ici le code
    }

    public function getPermissions(): array {
        // ici le code
    }
}
```

```
class User {
    use HasRoles, HasPermissions;

    // Le reste du code de la classe
}
```

Implémenter un trait "HasId" qui définit une propriété \$id et les méthodes setter et getter

```
spl_autoload_register(function($class_name) {  
    $file = __DIR__ . '/classes/' . $class_name . '.php';  
    if (file_exists($file)) {  
        require_once $file;  
    }  
});
```

- ✓ Une seule classe par fichier
- ✓ Le fichier doit porter le même nom que la classe

Sépare la gestion des erreurs
du code qui provoque l'erreur,
ce qui permet de :

- ✓ Différer la gestion des erreurs
- ✓ Permettre un traitement des erreurs différencié
- ✓ Simplifier le code en séparant les responsabilités

lancement d'une exception

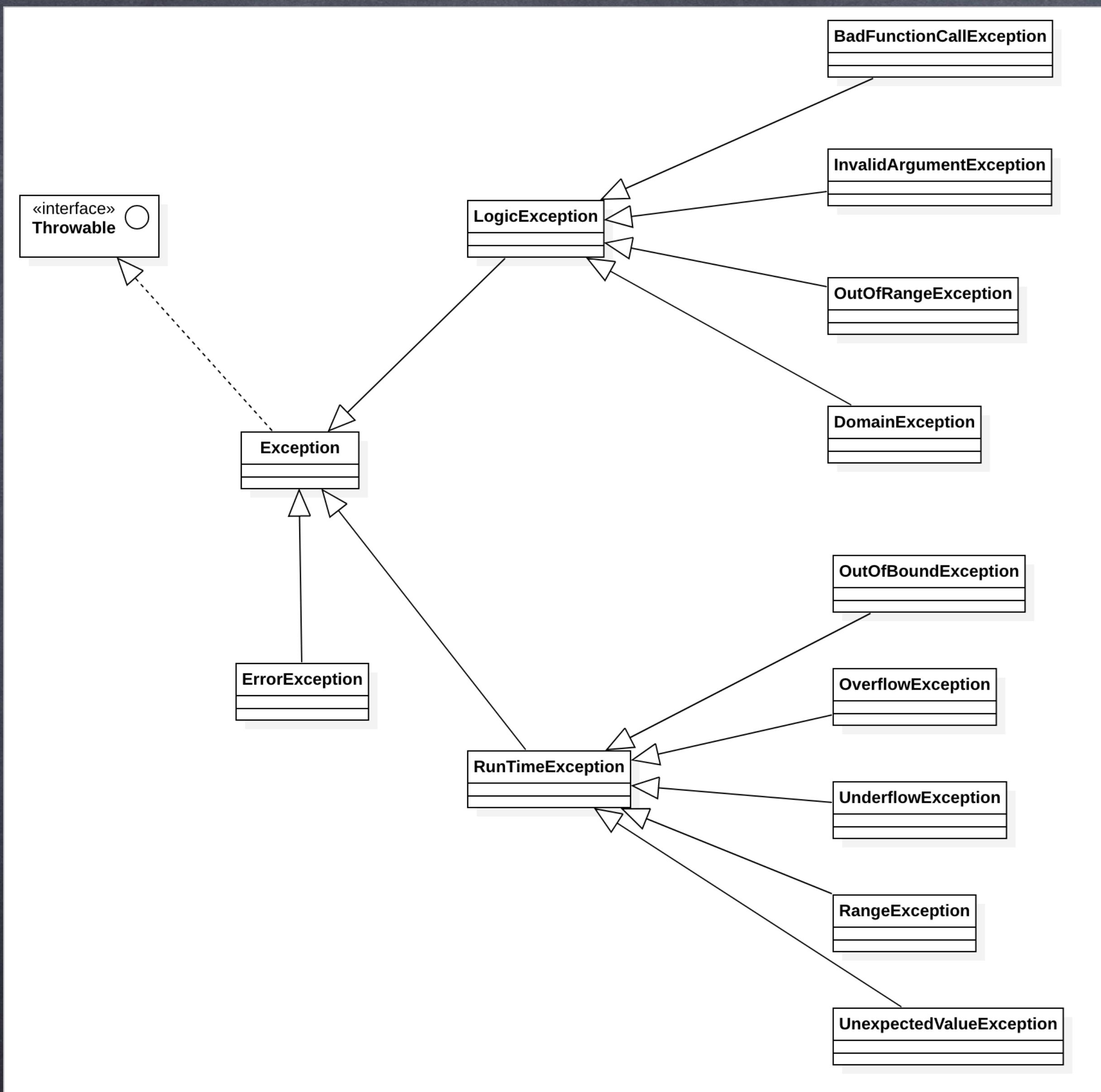
Exécution du code
dans un bloc try

Capture de l'exception

```
function inverse($x) {
    if (!$x) {
        throw new Exception('Division par zéro.');
    }
    return 1/$x;
}

try {
    echo inverse(5) . "\n";
    echo inverse(0) . "\n";
} catch (Exception $e) {
    echo 'Exception reçue : ', $e->getMessage(), "\n";
}
```

```
try {
    $pdo = new PDO('mysql:host=localhost;dbname=test', 'root', '');
    $pdo->query('SELECT * FROM test');
} catch (PDOException $e) {
    echo 'PDOException reçue : ', $e->getMessage(), "\n";
}
```



```
function inverse($x) {
    if (! is_numeric($x)) {
        throw new InvalidArgumentException('Argument is not numeric');
    }
    if ($x === 0) {
        throw new InvalidArgumentException('Argument is 0');
    }

    return 1/$x;
}

try {
    echo inverse(5) . "\n";
    echo inverse(0) . "\n";

} catch(InvalidArgumentException $e){
    echo 'InvalidArgumentException : ', $e->getMessage(), "\n";

} catch (Exception $e) {
    echo 'Exception reçue : ', $e->getMessage(), "\n";
}
```

Exécution de code à la fin d'un bloc try
que les opérations aient échouées ou non

```
function processFile($filename) {  
    $file = fopen($filename, 'r');  
    try {  
        // Traitement du fichier  
    } catch (Exception $e) {  
        // gestion de l'exception  
    } finally {  
        // fermeture du fichier  
        fclose($file);  
    }  
}
```

La gestion de dépendances



<https://getcomposer.org/download/>

- ✓ **Gestion des dépendances** : Composer télécharge et installe automatiquement les bibliothèques PHP requises pour un projet.
- ✓ **Versionning** : Il permet de spécifier les versions exactes des dépendances nécessaires, assurant ainsi la compatibilité entre les différents composants.
- ✓ **Autochargement** : Composer offre la possibilité aux paquets d'enregistrer un autochargeur, facilitant l'utilisation des classes dans le projet.

```
composer init
```

Initialisation du projet

```
composer require <lib>
```

Installation de la dépendance nommée lib

```
composer require --dev <lib>
```

Installation de la dépendance de dev nommée lib

```
composer install
```

Installation des dépendances déclarées dans composer.json

```
composer update
```

Mise à jour des dépendances déclarées dans composer.json

```
composer require mexitek/phpcolors
```

```
<?php
include 'vendor/autoload.php';

$backgroundColor = new Mexitek\PHPCOLORS\Color('#880000');
$foregroundColor = $backgroundColor->complementary();

echo $foregroundColor;
```

```
<?php
include 'vendor/autoload.php';

use Mexitek\PHPCOLORS\Color;

$backgroundColor = new Color('#880000');
$foregroundColor = $backgroundColor->complementary();

echo $foregroundColor;
```

Import de la classe, évite d'avoir à écrire le namespace complet

```
composer require --dev fakerphp/faker
```

```
<?php
require_once 'vendor/autoload.php';

$faker = Faker\Factory::create();

echo $faker->name(). '<br>';
echo $faker->email(). '<br>';
echo $faker->text(). '<br>';
```

Où trouver des projets

The screenshot shows a web browser window with the Packagist website loaded. The search bar at the top contains the text 'faker'. Below the search bar, there is a message: 'Packagist is the main Composer repository. It aggregates public PHP packages installable with Composer.' To the left of this message is a small icon of an elephant.

The main content area displays a list of packages:

- fakerphp/faker** (PHP) - 234 032 283 downloads, 3 732 stars. Description: Faker is a PHP library that generates fake data for you.
- fzaninotto/faker** (PHP) - 218 945 991 downloads, 26 719 stars. Description: Faker is a PHP library that generates fake data for you. Status: Abandoned!
- nelmio/alice** (PHP) - 35 867 391 downloads, 2 535 stars. Description: Expressive fixtures generator.
- zenstruck/foundry** (PHP) - 5 992 341 downloads, 690 stars. Description: A model factory library for creating expressive, auto-completable, on-demand dev/test fixtures with Symfony and Doctrine.
- theofidry/alice-data-fixtures** (PHP) - 22 603 015 downloads, 312 stars. Description: Nelmio alice extension to persist the loaded fixtures.

Répertoires virtuels qui facilitent les points suivants :

- ✓ Organisation du code : Ils permettent de structurer le code de manière logique, similaire à une arborescence de dossiers⁴.
- ✓ Évitement des conflits : Ils résolvent les problèmes de noms identiques entre différentes parties du code ou bibliothèques.
- ✓ Autoloading : Ils facilitent la mise en place d'un système d'autoloading des classes.
- ✓ Lisibilité : Ils améliorent la lisibilité du code en évitant les noms de classes trop longs ou préfixés.

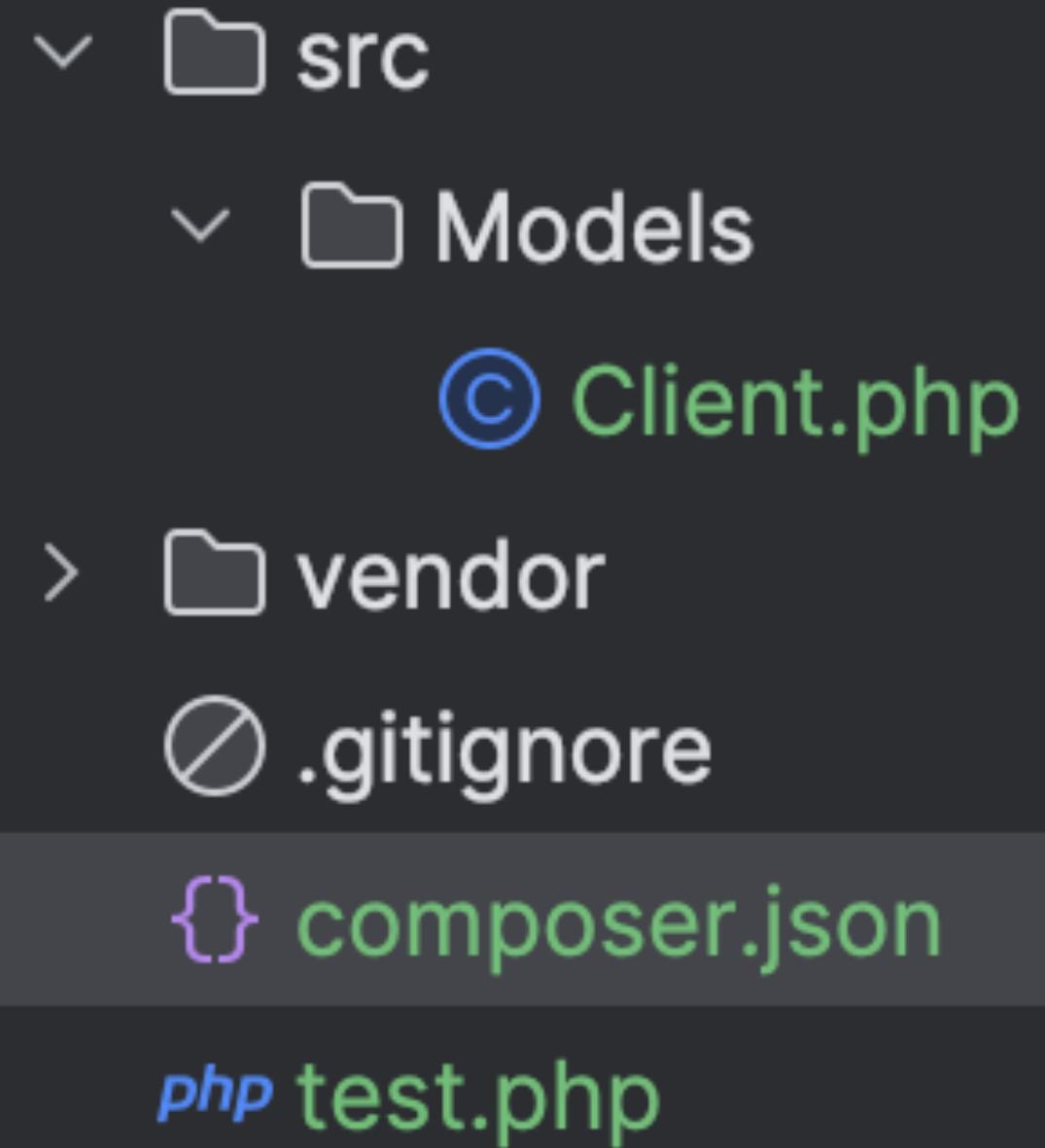
```
<?php

namespace Smaloron\Project\Model;

class Client
{
    public function __construct(
        private int $id,
        private string $name,
        private string $email
    )
    {}

}
```

```
composer dump-autoload
```



```
{
    "autoload": {
        "psr-4": {
            "Smaloron\\Project\\": "src"
        }
    }
}
```

```
<?php
include 'vendor/autoload.php';

use Smaloron\Project\Model\Client;

$client = new Client(5, "Jane", "Doe");
```

Applications

© QueryBuilder

© QueryBuilder

• fieldNames
• allowedVerbs
• verb
• limit
• orderBy
• whereClauses
• tableName

• limit(limit, offset)
• addOrderBy(fieldName, direction)
• insert()
• delete()
• setFields(fieldNames)
• update()
• from(tableName)
• addWhere(clause)
• addField(fieldName)
• getQuery()
• select(tableName)
• getORDERBYClause()
• getDELETEQuery()
• getSELECTQuery()
• getUPDATEQuery()
• getFieldList()
• getINSERTQuery()
• getWHEREClause()

```
$qb = new QueryBuilder();
$qb ->select()
->from('contacts')
->addField('name')
->addField('email')
->addWhere('id=?');
echo $qb->getQuery();
```

Construire la requête étapes par étapes
et ne générer le code sql qu'à la fin

© DAO

- Ⓜ __construct(pdo, queryBuilder, tableName)
- Ⓟ pdo
- Ⓟ queryBuilder
- Ⓟ tableName
- Ⓜ ⌂ find(whereClauses)
- Ⓜ ⌂ insert(data)
- Ⓜ ⌂ deleteOneById(id)
- Ⓜ ⌂ findOneById(id)
- Ⓜ ⌂ update(data, whereClauses)
- Ⓜ ⌂ findAll()
- Ⓜ ⌈ executeQuery(sql, params)
- Ⓜ ⌈ executeSelectQuery(sql, params, onlyOne)

```
include 'vendor/autoload.php';

use smaloron\models\DAO;
use smaloron\models\QueryBuilder;

$pdo = new PDO('mysql:host=localhost;dbname=test',
'root', '');

$qb = new QueryBuilder();

$personDao = new DAO($pdo, $qb, 'persons');

$allPersons = $personDao->findAll();
```

Une abstraction des requêtes à un SGBDR

© View

- ⊕ ⚡ layout
 - ⊕ ⚡ data
 - ❖ ⚡ viewFolder
 - ⊕ ⚡ content
-
- Ⓜ ⚡ render(template, data)
 - Ⓜ ⚡ renderSection(name)
 - Ⓜ ⚡ setLayout(layout)

```
<?php
include 'vendor/autoload.php';
use Smaloron\App\Core\View;

View::$viewFolder = __DIR__ . '/templates/';

$view = new View();

echo $view->render('test', [
    'name' => 'Sebastien',
    'title' => 'Test de la vue'
]);
```

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
        <title><?php $this->renderSection('title'); ?></title>

    <style>
        body{
            padding: 20px;
            background: beige;
        }
        #main {
            display: grid;
            grid-template-columns: 1fr 3fr;
            grid-gap: 20px;
        }
    </style>
</head>
<body>
    <div id="main">
        <nav>La navigation</nav>
        <div id="content">
            <?php $this->renderSection('content'); ?>
        </div>
    </div>
</body>
</html>
```

```
<h1>Test</h1>

Hello <?= $name ?>
```