Architecture(s) et application(s) Web

Télécom SudParis Olivier Berger (TSP)

01/09/2024

CSC4101 - Cours PHP

Table des matières

1 Généralités sur le langage PHP 3

2 Langage PHP 6

3 Syntaxe objet 8

4 Installation des outils et bibliothèques 12

5 Mettre au point et tester le code 17

Objectifs de cette séquence

L'objectif de cette séquence du cours est de présenter le premier grand ensemble de technologies de mises en œuvre qui vont nous servir dans les séquences pratiques du cours, autour du langage PHP.

1 Généralités sur le langage PHP

En introduction de cette séquence, voici, pour votre information quelques éléments relatifs aux choix des technologies du monde PHP qu'on emploiera dans ce module.

1.1 Survol rapide

— Pas un cours pour enseigner le langage : apprentissage en autonomie
— Quelques éléments pour planter le décor

Les bases du langage seront étudiées en travail autonome, sur des ressources externes.

1.2 Faire tourner des programmes sur le serveur Web

— Besoin : Programmer des applications (couche traitements, qui s'exécute côté serveur)
— Choix d'un **langage** ?
— Choix d'une **technologie associée** ?

Dans ce module, notre but est de pouvoir faire tourner une application sur un serveur Web.
De nombreux langages et environnement de développement Web sont disponibles.
Pour une découverte de la programmation des applications Web par la pratique, nous avons fait le choix du **langage PHP** et de l'**environnement Symfony**, pour leur qualités dans un contexte pédagogique.
Une grande partie des aspects techniques étudiés se retrouveront dans d'autres langages ou *frameworks*.

1.3 Langage choisi : PHP

— Langage nouveau (pour vous)
— « Proche » de Java (ou C)
— Interprété (comme *Bash* ou *Python*)

Mais aussi des outils…

1.4 Mais pourquoi tant de haine ?

Cf. « Why developers hate PHP » par Mehdi Zedi

— Vous aimez apprendre des choses à la mode ?
vous allez voir des choses modernes, *si si* (objet)
— PHP est un vieux langage à la mauvaise réputation
— Vous allez devoir bosser (de plus en plus) sur des vieux trucs : importance de la *maintenance* vs innovation (ou innovation dans la maintenance) : *ODD* ?
— Les outils qu'on vous présente (*Symfony*) sont à l'**état de l'art** (objet, génie logiciel)
— Approche pédagogique : **complexité maîtrisée** (application des concepts en 2A, chaque chose en son temps)

PHP est un vieux langage, et de nombreux tutoriaux ou manuels expliquent comment le prendre en main.
PHP est parfois dénigré car il supporte encore des façons de programmer qui sont déconseillées depuis longtemps.
Ce n'est pas une raison pour jeter PHP à la poubelle. Il y a de bonnes pratiques pour utiliser PHP de façon moderne et produire des programmes de bonne qualité.
Nous présentons rapidement quelques-uns de ces éléments, dans ce cours, que nous reverrons plus en détail dans les phases pratiques.

1.5 Symfony : PHP moderne

1.5.1 PHP « comme il faut »

S'appuyer sur un cadriciel (*framework)* moderne comme Symfony
- orienté objet
- fonctionnel
- injection de dépendances, conteneurs
- *templates*
- PHPDoc
- tests
- …

Suite : `http://www.phptherightway.com/`

La plupart des cours ou tutoriaux qu'on trouve sur PHP sur le Net datent un peu… pourtant il existe d'excellents documents actuels, comme le site ci-dessus « *PHP the right way* » de Josh Lockhart *et al.*

1.6 Le cadriciel moderne : Symfony 6

- *Framework* de référence
- Assemblage de beaucoup de bibliothèques
- Modèle de composants objet évolué
- Documentation
- Communauté
- **Environnement de mise au point**

`https://symfony.com/`

Dans le cours, on utilisera Symfony 6, la version « Long-Term Support (LTS) » actuelle.
Attention, la version 7, est la version *stable* actuelle, mais nous n'avons pas adapté les contenus pour l'utiliser.
Attention aux documents ou tutoriaux portant sur des versions antérieures, qui foisonnent sur le Web, et ne s'appliquent pas forcément à cette version récente de Symfony.

1.6.1 Une *licorne* française - SensioLabs

`https://sensiolabs.com/`

SensioLabs

GOUVERNEMENT.fr ”HistoiresdeFrance, chapitre 21 (2016)”

« *SensioLabs, une des plus belles réussites françaises du web* » –
In #HistoiresdeFrance, chapitre 21 (*post X du Gouvernement Français*, décembre 2016 - cf. sauvegarde sur/ Internet Archive)

1.7 Autres frameworks :

PHP Laravel, CodeIgniter, CakePHP, Zend, …

Ruby Ruby on Rails, Sinatra, …

Node.js Meteor, Express.js, …

Python Django, Flask, …

Java Spring, Struts,

2 Langage PHP

L'apprentissage des bases du langage sera fait en autonomie dans la prochaine séquence hors-présentielle.
Nous allons présenter ici quelques aspects avancés du langage et de son environnement, non couverts par les tutoriels de base, et qui serviront dans la suite du cours dès la prochaine séquence de travaux pratiques.
Il n'est pas utile de comprendre tous les détails dès maintenant, mais de pouvoir s'y référer ultérieurement.

2.1 Langage

- Syntaxe style C / Java
- **Objets**
- Interprété
- Héritage contexte Web, CGI (« *PHP : Hypertext Preprocessor* »)
- Depuis 1994 (PHP 8 depuis fin 2020)
- Versions pour ce cours : >= 8.3

PHP est l'un des langages les plus populaires sur le Web. Cf. Usage statistics and market share of PHP for websites pour quelques éléments chiffrés.
Les versions de PHP que vous allez utiliser devraient typiquement être PHP 8, comme sur machines DISI salle de TP
Pour plus de détails sur les *elePHPants* (mascote du langage), voir Comment et pourquoi la mascotte PHP est-elle venue à la naissance ? L'histoire secrète d'ElePHPant ! et A Field Guide to Elephpants - Detailing the attributes, habitats, and variations of the Elephpas hypertextus.

2.1.1 *Hello world*

```
<html>
  <head>
    <title>Test PHP</title>
  </head>
  <body>
    <?php echo '<p>Bonjour le monde</p>'; ?>
  </body>
</html>
```

ou bien :

```
<?php

  $html = "<html><head><title>Test PHP</title></head><body>";
  $html .= "<p>Bonjour le monde</p>";
```

```php
$html .= "</body></html>";
print($html);
```

PHP permet d'écrire des morceaux de programme à l'intérieur de pages HTML (*inline*).
Que peut-on en penser ?

2.1.2 PHP *inline*

— Mélanger la présentation (HTML) et le code (PHP)… **c'est mal** : maintenable ?
— On verra comment faire autrement dans une prochaine séquence.

Cf. les gabarits (*templates*) en séquence 3.

2.2 La documentation

— Le site de PHP : `http://php.net/`
— La documentation : `http://php.net/manual/fr/`

La documentation de référence est à préférer, en cas de doute, notamment pour les fonctions de la bibliothèque standard.
Cependant, on peut douter de sa qualité pédagogique pour l'apprentissage.
On verra que la documentation de Symfony est par contre d'excellente qualité en général.

2.3 Syntaxe

Vous allez commencer à l'apprendre en hors-présentiel, cette semaine
Vous devriez comprendre ce que je vais montrer

3 Syntaxe objet

Ce cours ne contient pas de manuel d'introduction à la programmation en PHP. Nous présentons ici quelques éléments particulièrement importants du modèle objet, et des fonctionnalités avancées utiles dans un projet Symfony.

3.1 Appel de méthode

```php
// crée une instance de la classe SymfonyStyle

$io = new SymfonyStyle($input, $output);

// appel de la méthode title() sur cette instance
// pas de valeur de retour

$io->title('list of todos:');
```

```php
namespace Symfony\Component\Console\Style;
/**
 * Output decorator helpers for the Symfony Style Guide.
 */
class SymfonyStyle extends OutputStyle
{

// [...]
```

```php
namespace Symfony\Component\Console\Style;
/**
 * Decorates output to add console style guide helpers.
 */
abstract class OutputStyle implements OutputInterface, StyleInterface
{

// [...]
```

```php
namespace Symfony\Component\Console\Style;
/**
 * Output style helpers.
 */
interface StyleInterface
{
    /**
     * Formats a command title.
     */
    public function title(string $message);
```

— Syntaxe java : `instance.methode()`
— Syntaxe PHP : `$instance->methode()`

3.2 Constructeur, propriétés internes

```php
// Classe

class MyCommand
{
        // Propriété / attribut / variable d'instance

        private $todoRepository;

        // Constructeur

        public function __construct($repository)
```

```
        {
                $this->todoRepository = $repository;
        }

        // Méthode d'instance

        protected function execute()
        {
                $todos = $this->todoRepository->findAll();
        }
```

— Syntaxe java : `this.propriete`
— Syntaxe PHP : `$this->propriete`

3.3 Valeur de retour

```
class ListTodosCommand extends Command
{
    // [...]
    protected function execute(SymfonyStyle $io): int
    {
        $todos = $this->todoRepository->findAll();

        if( empty($todos) ) {
                return Command::FAILURE;
        }
        else {
                $io->title('list of todos:');
                $io->listing($todos);
        }

        return Command::SUCCESS;
    }
```

— `Command::SUCCESS` et `Command::FAILURE` sont des propriétés statiques de la classe `Command` (des constantes)
— `empty()` : tableau vide. Mais en fait « *A variable is considered empty if it does not exist or if its value equals false.* » (Cf. empty — Determine whether a variable is empty)

3.4 Surcharge / héritage

```
use Symfony\Component\Console\Command\Command;

// Classe ListTodo hérite de Command

class ListTodosCommand extends Command
{
        // Constructeur

        public function __construct($repository)
        {
                $this->todoRepository = $repository;

                // Appel du constructeur de la classe mère

                parent::__construct();
        }

        // [...]
```

— Hérite d'une classe existante de la bibliothèque Symfony
— Surcharge du comportement par défaut dans le constructeur, donc appel au constructeur de la classe parente (ici `Command`)

3.5 Espaces de noms

```php
namespace App\Command;

use Symfony\Component\Console\Command\Command;
use App\Entity\Todo;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;

// Classe App\Command\MyCommand

class MyCommand extends Command
{
        // ...
        public function __construct(ManagerRegistry $manager)
        {
                $this->todoRepository = $manager->getRepository(Todo::class);

                parent::__construct();
        }

        protected function execute(InputInterface $input, OutputInterface $output): int
        {
                // ...
                return Command::SUCCESS;
        }
```

— `Todo::class` : l'objet classe lui-même (appel de méthodes statiques, etc.). Pas une instance de cette classe.

3.6 *Docblocks* PHPDoc

— Commentaires améliorés
— méta-informations

```php
/**
 * Classe "Circuit" du Modèle
 */
class Circuit
{
...
    /**
     * Set description
     *
     * @param string $description
     *
     * @return Circuit
     */
    public function setDescription($description)
    {
        ...
```

Les *Docblocks* PHPDoc étendent le langage PHP en fournissant le moyen d'ajouter des méta-informations dans des « commentaires améliorés » associés au code PHP.
Ils sont particulièrement utiles pour embarquer la documentation des entités appelées, pour s'y référer quand on programme les entités appelantes : la documentation est accessible dans l'éditeur/IDE, qui peut proposer des fonctions d'auto-complétion et de détection des erreurs dans le code.
C'est particulièrement utile quand on est dans un langage interprété (pas de compilateur pour éviter certaines erreurs), et quand on n'a pas le temps de lire les manuels des bibliothèques !
Par exemple, dans l'exemple ci-dessus, l'éditeur proposera directement la saisie d'une chaîne de caractères quand le programmeur saisit une invocation de la méthode `setDescription()`.
Ce mécanisme et sa syntaxe sont identiques à d'autres langages de programmation, avec par exemple JavaDoc en Java.
Cf. documentation de PHPDocumentor : `https://docs.phpdoc.org/` pour plus de détails.

3.7 Annotations : attributs PHP (> PHP 8.x)

```php
// AsCommand: "Service tag to autoconfigure commands"
use Symfony\Component\Console\Attribute\AsCommand;

use Symfony\Component\Console\Command\Command;

// the command name and description shown when running "php bin/console list"
#[AsCommand(
    name: 'app:list-todos',
    description: 'List tasks',
)]
class ListTodosCommand extends Command
{
```

façon déclarative d'enrichir le code, au lieu de :

```php
use Symfony\Component\Console\Command\Command;

class ListTodosCommand extends Command
{
    // the command name and description shown when running "php bin/console list"
    protected static $defaultName = 'app:list-todos';
    protected static $defaultDescription = 'List tasks';
```

Les attributs ajoutent des méta-données dans le code source PHP, et permettent une programmation réflexive (introspection).
On « décore » le code avec des propriétés utiles, par exemple des contraintes sur la validité, ou le stockage des données.
On verra que le composant Doctrine, ou le routeur Symfony utilisent ces attributs pour « décorer » le code, facilitant la maintenance en ne dispersant pas à différents endroits les propriétés de l'application.
Il est préférable d'utiliser un éditeur récent compatible (comme *Eclipse PDT* en version supérieure à 2023-06), pour faciliter la reconnaissance de ces attributs et leurs ajouts dans le code.

4 Installation des outils et bibliothèques

Nous installerons/utiliserons la variante CLI (*Command Line Interface*)
- **Interpréteur ligne de commande** (`php-cli`)

```
$ php helloworld.php
```

- ~~Interpréteur invoqué par le serveur HTTP (`php`)~~
- Bibliothèques :
  - `php-sqlite3`
  - `php-intl`
  - `php-xml`
  - …

La distribution PHP peut être installée en deux variantes selon le contexte d'utilisation.
Attention à bien spécifier la bonne variante à l'installation.
Dans le cours, nous utiliserons principalement la première, pour le développeur PHP : pour la ligne de commande (installer le paquetage `php-cli`).
La seconde est plutôt utilisée pour la mise en production sur des serveurs Web.

4.1 Bibliothèques complémentaires

Logiciels faisant partie de l'écosystème de bibliothèques, composants, *frameworks* PHP.
Sources :
- **Composer** : utiliser des bibliothèques libres
- ~~PEAR (*PHP Extension and Application Repository*)~~

Les bibliothèques distribuées via PEAR sont typiquement déjà installées (précompilées) via des paquetages de distributions GNU/Linux. Elles ne sous intéresseront pas dans le contexte de ce cours.
On va par-contre utiliser *Composer*, dont voici quelques caractéristiques.

4.2 Composer

Gestionnaire de paquetages PHP.

`https://getcomposer.org/`
Il fournit :
- gestionnaire de dépendances entre bibliothèques (et entre versions),
- téléchargement des paquetages des bibliothèques depuis *packagist*
- *autoloader* qui facilite les déclarations et les chargements associés au démarrage des programmes,

Composer est le gestionnaire de paquetages PHP, comme il en existe pour de nombreux langages de programmation.
On va utiliser Composer intensément quand on développera avec le *framework* Symfony. Les bibliothèques PHP utiles à un projet sont téléchargées grâce à Composer et installées dans le répertoire du projet.

4.2.1 Packagist (optionnel)

`https://packagist.org/`
Référentiel en ligne de paquetages (*packages*) pour Composer :
- contributif
- différentes versions de chaque paquetage

Nous mentionnons l'existence de *Packagist* uniquement pour votre culture. On s'en servira via Composer, de façon transparente.
Packagist constitue un dépôt de paquetages contributif, où des développeurs PHP peuvent publier leurs bibliothèques (libres).
N'importe qui peut y publier ses contributions.

Packages registered	374 425
Versions available	4 008 934
Packages installed	84 730 075 331

(since 2012-04-13)
(source : `https://packagist.org/statistics`, au 28/06/2023) Une fois qu'un paquetage est référencé sur packagist, il est téléchargeable par d'autres développeurs PHP grâce à Composer.
Pour publier un paquetage, il suffit par exemple de rendre public un référentiel Git, ce qui facilite la contribution de paquetages PHP, en logiciel libre, par exemple depuis GitHub.

4.2.2 Descripteur `composer.json`

Exemple de description d'un *projet local* :

```
{
    "type": "project",
    "require": {
        "php": ">=8.1",
        "symfony/console": "^6.4",
        "symfony/flex": "^1.0",
        "symfony/framework-bundle": "^6.4"
    },
    "require-dev": {
        "symfony/dotenv": "^6.4"
    },
    "config": {
        "preferred-install": {
            "*": "dist"
        },
        "sort-packages": true
    },
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    }
}
```

Chaque projet contiendra donc un fichier `composer.json` dont le contenu est renseigné par les développeurs pour y définir les dépendances particulières à télécharger.
Cet exemple (fictif) définit l'environnement nécessaire au développement d'une application Symfony.
On trouve les règles suivantes :

- dépendance sur PHP 8
- dépendance des bibliothèques nécessaires à l'exécution de l'application déployée : `symfony/console`, `symfony/flex` et `symfony/framework-bundle` dans leurs versions respectives (pour Symfony 6.4)
- dépendance sur la bibliothèque `symfony/dotenv` pour l'environnement de développement
- configuration de l'auto-loader pour associer l'espace de noms `App` au contenu du sous-répertoire `src/` de l'application (les codes source PHP écrites par le développeur s'y trouveront).
- etc.

En général, on s'inspire d'une documentation, ou on utilise un générateur, pour ne pas avoir à comprendre tous ces détails.
Voyons maintenant comment on l'utilise.

4.2.3 Installation des bibliothèques

1. Le développeur lance :

```
$ composer install
```

2. Analyse des règles contenues dans `composer.json`
3. Résultat est téléchargé dans `vendor/`

```
autoload.php
autoload_runtime.php
bin/
composer/
doctrine/
easycorp/
friendsofphp/
laminas/
masterminds/
monolog/
nikic/
psr/
symfony/
twig/
```

```
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 20 installs, 0 updates, 0 removals
  - Installing symfony/flex (v1.0.89): Downloading (100%)
Enable the "cURL" PHP extension for faster downloads

Prefetching 19 packages
  - Downloading (100%)

  - Installing symfony/polyfill-mbstring (v1.9.0): Loading from cache
  - Installing symfony/console (v4.1.3): Loading from cache
  - Installing symfony/routing (v4.1.3): Loading from cache
  - Installing symfony/polyfill-ctype (v1.9.0): Loading from cache
  - Installing symfony/http-foundation (v4.1.3): Loading from cache
  - Installing symfony/event-dispatcher (v4.1.3): Loading from cache
  - Installing psr/log (1.0.2): Loading from cache
  - Installing symfony/debug (v4.1.3): Loading from cache
  - Installing symfony/http-kernel (v4.1.3): Loading from cache
  - Installing symfony/finder (v4.1.3): Loading from cache
  - Installing symfony/filesystem (v4.1.3): Loading from cache
  - Installing psr/container (1.0.0): Loading from cache
  - Installing symfony/dependency-injection (v4.1.3): Loading from cache
  - Installing symfony/config (v4.1.3): Loading from cache
  - Installing psr/simple-cache (1.0.1): Loading from cache
  - Installing psr/cache (1.0.1): Loading from cache
  - Installing symfony/cache (v4.1.3): Loading from cache
  - Installing symfony/framework-bundle (v4.1.3): Loading from cache
  - Installing symfony/dotenv (v4.1.3): Loading from cache
Writing lock file
Generating autoload files
```

Composer construit le graphe transitif des dépendance en partant des 4 dépendances `symfony/console`, `symfony/flex`, `symfony/framework-bundle` et `symfony/dotenv` qui ont été mentionnées explicitement dans `composer.json`, et télécharge les 20 paquetages qui en résultent (dépendances de base du noyau du *framework* Symfony).
Le code des différentes bibliothèques téléchargées est alors extrait dans des sous-répertoires du répertoire `vendor/` à la racine du projet.
L'extension Flex pour Composer (`symfony/flex`) est alors mise à contribution pour générer certains fichiers utiles au projet (cf. `https://github.com/symfony/flex`), si nécessaire (renseigner des valeurs par défaut, etc.).
Composer comporte bien d'autres fonctions, dont certaines seront utilisées plus tard, comme la création d'un squelette d'application, avec `composer create-project`.
Maintenant que tout le code des bibliothèque est présent, voyons comment il est chargé à l'exécution.

4.3 Chargement des bibliothèques via l'*autoloader (optionnel)*

Voici le fonctionnement du chargement via l'*auto-loader* lié à *Composer*, pour votre culture.
En pratique tout cela à transparent.
Dans l'exemple ci-dessous, le programme PHP `index.php` utilise l'*auto-loader* pour charger les bibliothèques nécessaires à son exécution.
Voici une bibliothèque dont le code est installé dans `vendor/symfony/http-foundation/Request.php` :

```php
namespace Symfony\Component\HttpFoundation;

...
```

```
    class Request {
```

et qui est chargée « automatiquement » via ce programme `index.php` :

```
<?php

require __DIR__.'/vendor/autoload.php';

...

$request = Symfony\Component\HttpFoundation\Request::createFromGlobals();
```

Ainsi lorsqu'il utilise une classe comme `Symfony\Component\HttpFoundation\Request`, celle-ci est trouvée par l'interpréteur, car présente dans l'espace de noms `Symfony\Component\HttpFoundation` que l'*auto-loader* aura trouvé dans la déclaration d'espace de noms présente dans le fichier source `vendor/symfony/http-foundation/Request.php`.
Les programmes utilisent des identifiants d'espaces de noms PHP et non le chemin en dur des fichiers sources, ce qui rend le code maintenable en permettant la restructuration de l'arbre des fichiers sources des bibliothèques.
Cf. `https://www.phptherightway.com/#namespaces`.

5 Mettre au point et tester le code

- Langage interprété
- Trouver les bugs avant exécution ?
- Tester (systématiquement)
- IDE (*Integrated Development Environment*), pour détecter les erreurs quand on tape le code

Les tests sont particulièrement importants dans un langage interprété comme PHP.
Avec un langage compilé (comme Java) un grand nombre de problèmes et de bugs sont identifiés lors de la phase de compilation et doivent être résolus assez tôt.
Avec un langage interprété, c'est uniquement quand le programme fonctionne que ceux-ci seront détectés. Mieux vaudrait que ça soit en phase de mise au point, quand un développeur compétent est disponible, plutôt qu'une fois en production.
Il est aussi intéressant d'utiliser un éditeur ou un environnement de développement intégré offrant un support du langage, pour identifier certaines erreurs au plus tôt.

5.1 Mise au point : affichage

Afficher des traces d'exécution
- `print()` est votre ami ?
- mais on peut faire mieux !

La programmation avec PHP est souvent très itérative et nécessite de maîtriser les bonnes pratiques pour la mise au point.
Premier outil pour la mise au point : afficher des traces.
Astuces « débug » sur sortie standard :
- `echo` / `print( )` : basique, pourvu qu'il y ait une sérialisation en chaîne de caractères
- `print_r( )` : affichage formaté

```php
echo '<pre>';
print_r($data);
echo '</pre>';
```

- `var_dump( )` : très détaillé… MAIS **attention aux récursions** !

Attention, les affichages ne sont pas toujours visibles : capture de la sortie standard, formatage des réponses HTTP : en-têtes / corps de réponse…
Une astuce pour capturer le formattage de `print_r` dans une variable :

```php
$affichage = print_r($data, 1);
```

5.2 `dump()` dans Symfony

Utiliser `dump()`

```php
dump($myarray);
```

```
array:5 [▼
  "a simple string" => "in an array of 5 elements"
  "a float" => 1.0
  "an integer" => 1
  "a boolean" => true
  "an empty array" => []
]
```

Dans les *frameworks* comme Symfony on verra des outils permettant de mettre au point plus confortablement, sans ces inconvénients, comme l'utilisation de la fonction `dump( )`.
Symfony nous apportera des fonctionnalités d'environnement de développement ou tests.

5.3 Environnement de développement local dans Symfony

Ainsi, on utilisera au quotidien, pour la mise au point un serveur Web local.
Il permet de tester le code en direct, en faisant un rechargement automatique des fichiers modifiés.
Il s'accompagera de l'utilisation d'une base de données locale (*SQLite*) permettant de tester complètement le fonctionnement de l'application sur l'ordinateur du développeur.

5.4 Tests PHPUnit

- environnement de tests : `PHPUnit`
- systématiser les tests :
  - conformité
  - non-régression
- Tester avant de déployer en production

Ne sera pas utilisé dans le cours (faute de temps)

Avoir une démarche de tests bien définie est indispensable Langage **interprété**
La meilleure technique pour assurer que le code est testé, est de l'accompagner d'une suite de tests automatisée, qui rend explicites les tests du programme.
Nous n'aborderons pas le domaine des tests dans le présent cours faute de temps, même si les *frameworks* comme Symfony apportent un support pour les tests assez élaboré.
En PHP, c'est l'environnement **PHPUnit** qui est utilisé. Il fonctionne de façon très similaire à JUnit de Java, par exemple.
Le sujet des tests sera abordé en détail dans le module optionnel CSC 4102 « Introduction au Génie Logiciel Orienté Objets ».

Take away

— PHP moderne
— Syntaxe objet
— Outils du développeur : *Composer*, `dump()`

Copyright