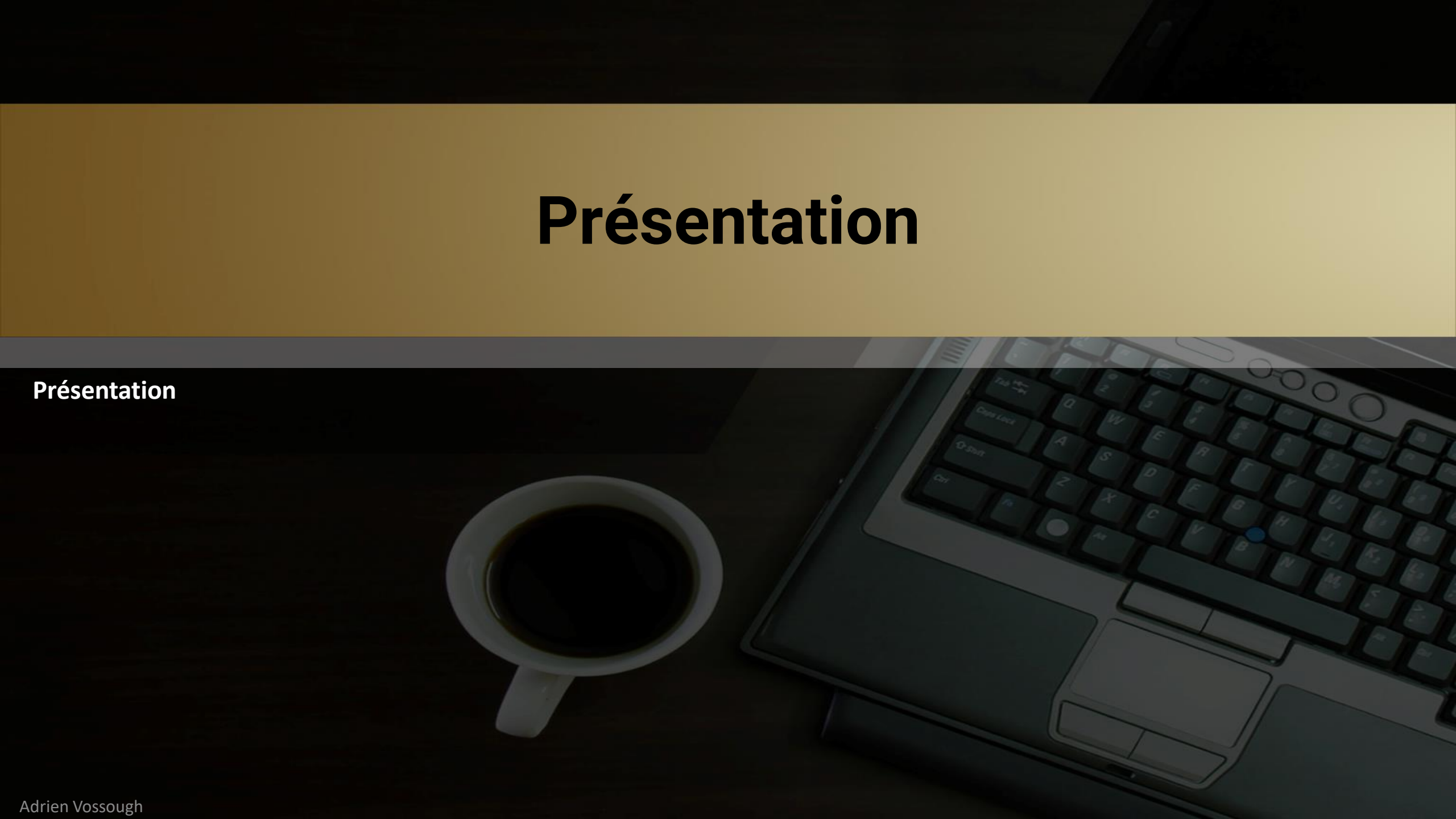
Adrien Vossough

Présentation



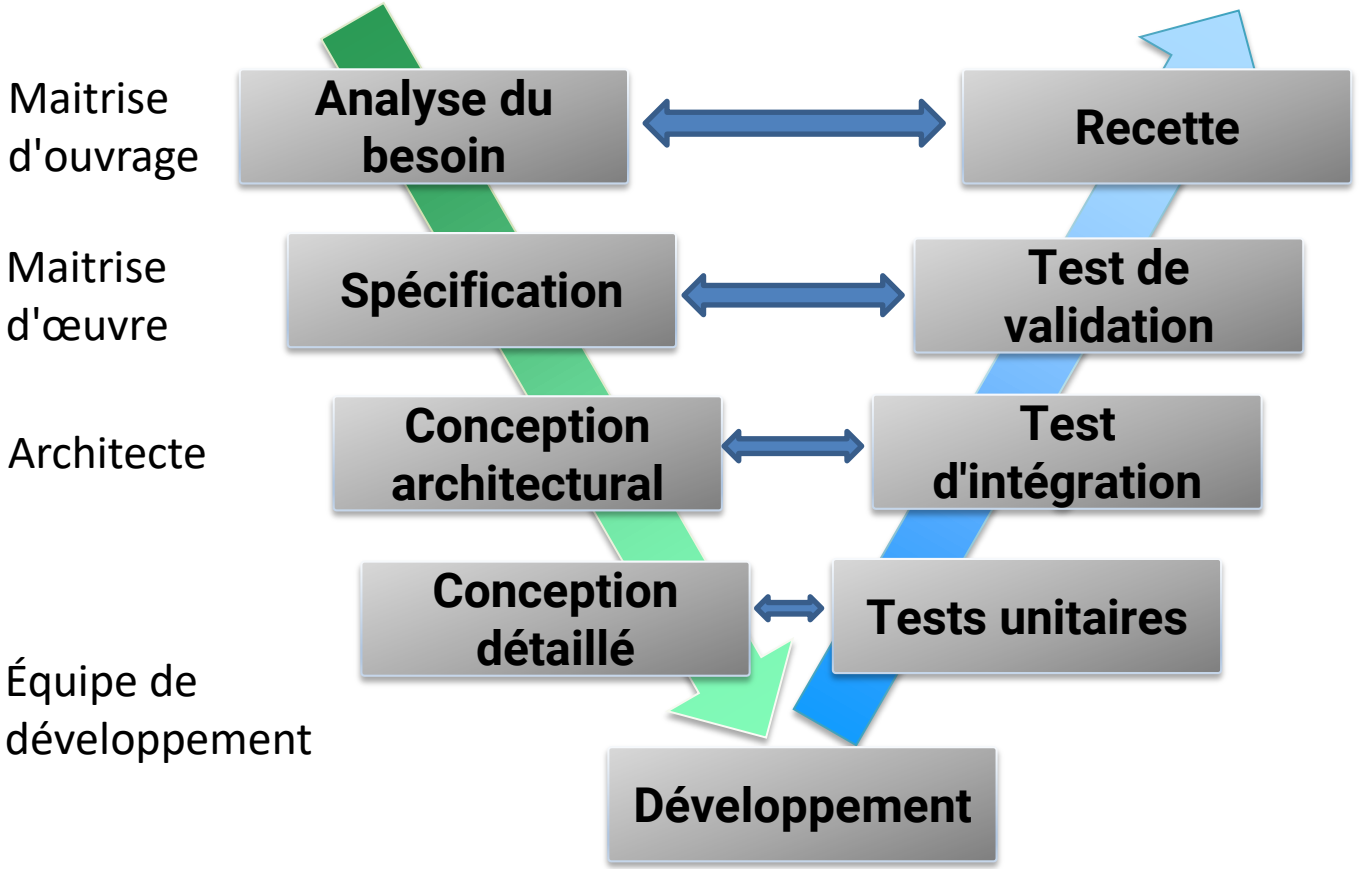
Présentation

Adrien Vossough

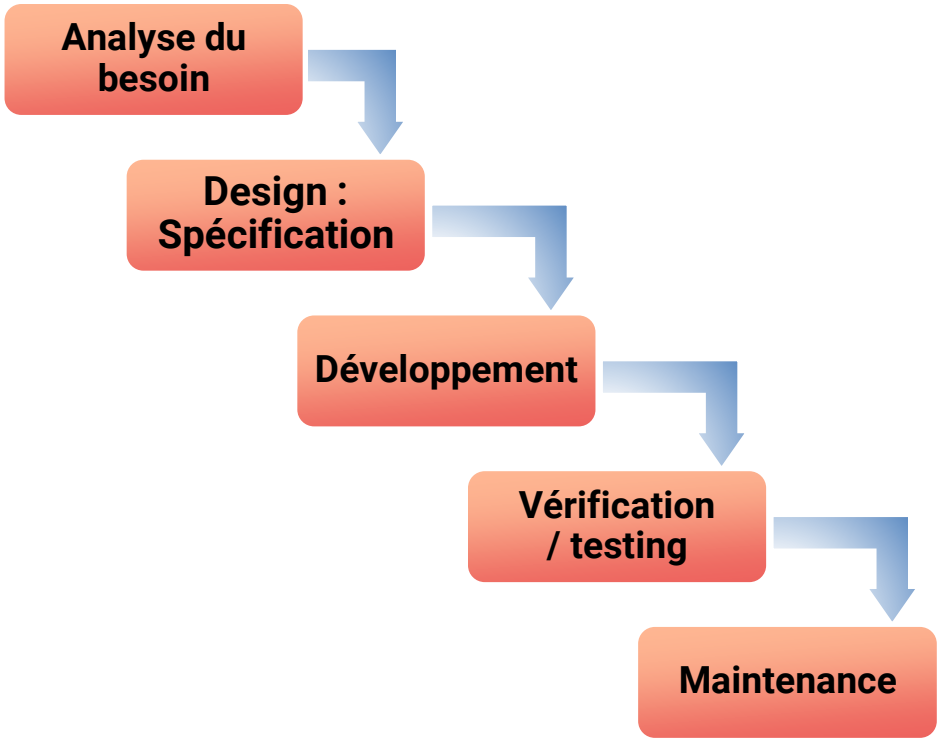
Rappel Agile/Scrum

L'Agile s'oppose aux approches prédictives et séquentielles de type cycle en V ou waterfall

Cycle en V



Waterfall



La conception en cycle en V ou Waterfall est faite en amont, avant le développement du produit et ne permet pas de s'adapter facilement au changement.

Le client n'a accès au produit qu'en phase de recette et ne peut donc pas suivre le processus de développement.

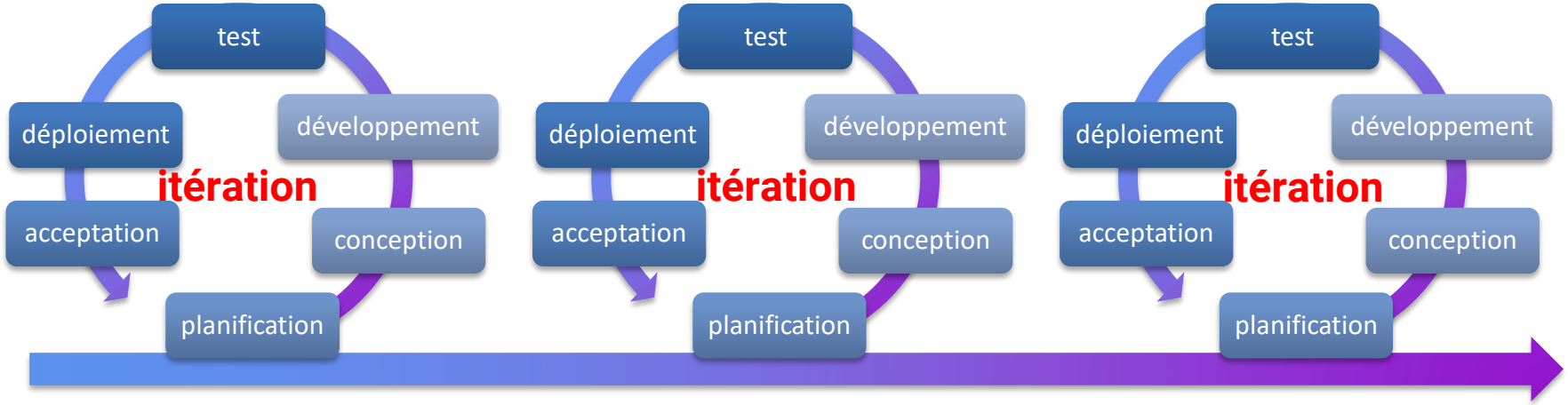
Si le produit est mal défini lors du cahier des charges, il ne sera pas en phase avec le besoin

Les méthodes agiles partent du principe que spécifier et planifier dans les détails l'intégralité d'un produit avant de le développer (approche prédictive) est contre-productif car beaucoup de changement peuvent être fait en route.

Elle se base sur une approche empirique et des objectifs à courts termes.

Le client présente sa vision du produit et ses exigences (fonctionnalités) à l'équipe de développement qui estime le coût de chacun des éléments.

Chacune des fonctionnalités sera développée sur un laps de temps appelé **sprint**/ **itération**



Chaque itération inclut des travaux de : conception, de spécification fonctionnelle et technique quand c'est nécessaire, de développement et de test.

A la fin de chacune de ces itérations, le produit partiel est montré au client qui valide les fonctions attendues et faire un retour

L'agile est un ensemble de méthodes de **travail basées sur 4 valeurs** fondamentales

- **Les individus et leurs interactions** plus que les processus et les outils
- **Des logiciels opérationnels** plus qu'une documentation exhaustive
- **La collaboration avec les clients** plus que la négociation contractuelle
- **L'adaptation au changement** plus que le suivi d'un plan

A cela, s'ajoute la phrase :
Nous reconnaissons la valeur des seconds éléments, mais privilégions les premiers.

De ces quatre valeurs sont nées les 12 principes du manifeste Agile

Les 12 principes du manifeste Agile

1. Satisfaire le client en priorité
2. S'adapter aux demandes de changement
3. Livrer fréq uemment un logiciel opérationnel avec des cycles courts.
4. Coopération permanente entre le client et l'équipe projet
5. Des personnes motivées avec un environnement et un soutien adapté
6. Privilégier la conversation en face à face
7. Mesurer l’avancement du projet en termes de fonctionnalités de l’application
8. Faire avancer le projet à un rythme soutenable et constant
9. Porter une attention continue à l’excellence technique et à la conception
10. Faire simple
11. Responsabiliser les équipes par l'architectures, les spécifications et conceptions
12. L'équipe chercher à s'adapter pour devenir plus efficacité

Parmi les méthodes agiles, le Scrum avec l'Extreme Programming est aujourd'hui la plus connue.

Scrum définit 3 rôles :

- Le « Product Owner » qui porte la vision du produit à réaliser (représente généralement le client).
- Le « Scrum Master » garant de la mise en application de la méthodologie Scrum.
- L'équipe de développement

Un projet Scrum est rythmée par un ensemble de réunions clairement définies et limitées dans le temps (timeboxing):

- **Planification du Sprint (itération)** : l'équipe sélectionne les éléments prioritaires du « Product Backlog » (exigences du projet) à réaliser au cours du sprint (en accord avec le « Product Owner »).

- **Revue de Sprint** : Réunion de fin du sprint pour lister les fonctionnalités terminées et recueille les feedbacks du Product Owner et des utilisateurs finaux. Elle permet aussi de préparer les prochains sprints

- **Rétrospective de Sprint** : Généralement après la revue de sprint, permet de faire une recherche sur les axes d'amélioration pour les futurs sprints

- **Mêlée quotidienne** : Mise au point sur l'avancement de chacun. En 15min, debout

Devops

Le développeur fabrique l'application tandis que **l'opérationnel** gère la mise en production et son exploitation.

L’équipe **Dev** fait évoluer les applications le plus rapidement possible pour réduire le time to market (délai entre l'idée et sa mise en production)

L'équipe **Ops** doit garantir la stabilité du système en passant par un contrôle sévère des changements apportés au Système d’information et ainsi éviter les pannes en production.

Le Devops est un mouvement né en 2009 qui vise l'unification du développement logiciel (dev) et de l'administration des infrastructures informatiques (ops).

Le but est d'aligner leurs objectifs court terme différents autour d'un objectif commun long terme : celui de la création de valeur pour l'entreprise.

Et cette création de valeur passe par **3 canaux:**

- Un time to market plus rapide
- Des produits de qualité
- Des équipes plus efficaces

Ce mouvement est basé sur **5 principes** de base :

- La collaboration
- La création de valeur ajoutée comme objectifs
- L’automatisation
- La mise en place de métriques à tous les niveaux
- Le partage comme valeur forte

Culture de collaboration :

- donner aux équipes une vision globale du système d'information.
- fluidifier et faciliter l'intercommunication entre développeurs et opérationnels.

La recherche de valeur ajoutée pour client :

- Mise en place d'un cycle d'amélioration continue et suppression des taches sans valeur ajoutée.
- Inciter les équipes à se concentrer sur la création de valeur business et la satisfaction client.

La mise en place de métriques :

- Amélioration par la mesure
- Métriques, métriques d'équipe et la mesure de la satisfaction client

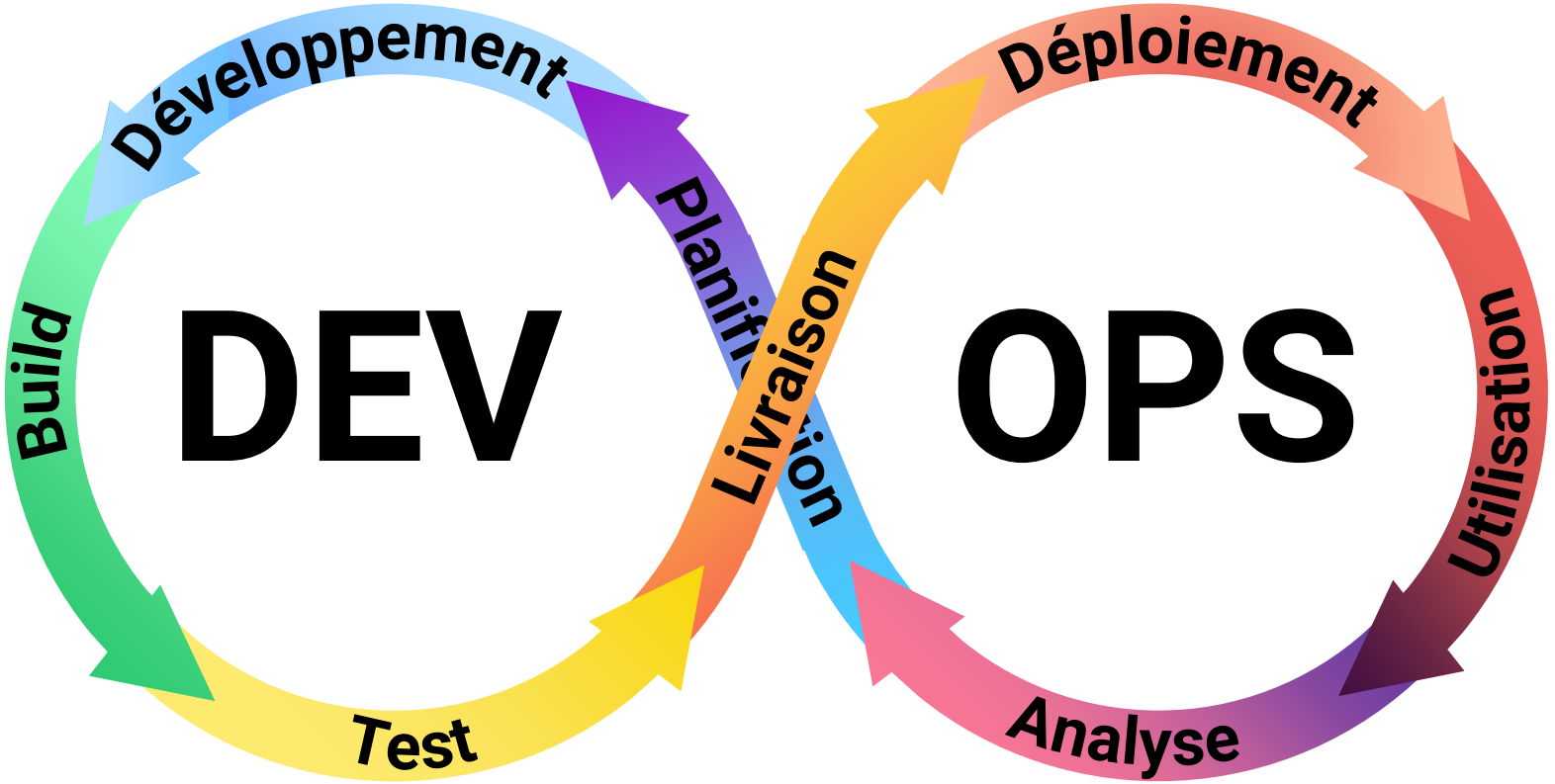
Principe de partage :

- Il s’agit de partager un objectif commun, mais aussi les problèmes, les connaissances et le retour d'expérience (REX).
- Faire émerger un sentiment d’entraide mais aussi de nouvelles idées.

Automatisation des processus :

  - Provisionning d’environnements automatisé pour les développeurs
  - Tests automatisés pour valider la qualité
  - Déploiements automatisés une fois le code validé et conforme aux attentes
- ***Aller vers le déploiement continu :***
  - Le déploiement continu considère qu’une application doit être faite de manière à partir en production à tout moment (après validation par les tests)
  - Permet d'accélérer les cycles de mise en production et par la même occasion le time to market.

Cycle de vie d'un projet.



Software Craftsmanship

Le software Craftsmanship est une approche agile axée sur la qualité technique et la réalisation.

Ce mouvement est apparu suite à l'accélération de la production des applications au détriment de la qualité.

Le mouvement Craftsmanship qui veut dire artisanat, opposé à "industriel", fait référence à un produit de bonnes qualités.

L'application doit :

- fonctionner
- être correctement écrit
- être fiable
- être maintenable.

Le craftmanship est basé sur des techniques de développement, de tests, de design, etc.

Manifeste du Software Craftsmanship

- Des logiciels opérationnels **ET** bien conçus.
- S'adapter au changement **ET** ajouter constamment de la valeur.
- Communiquer avec son équipe **ET** apprendre de la communauté professionnelle.
- Collaborer avec les clients **ET** avec des partenaires productifs.

L'amélioration de chacun passe par l'échange, les retours d'expériences, des jeux (confrontations de code)

Pour éviter la sur-conception/développement, un encadrement avec des développeurs expérimentés est important.

Les règles d'un artisan :

- Qualité du code (conception simple, tests, TDD)
- Humilité (se remettre en question et s'améliorer)
- Partage son expérience, ces acquis
- Pragmatisme (comprendre et s'adapter)
- Professionnalisme (le client est un partenaire à qui on peut dire non)
- Concret (OO, SOLID, DDD)

Règles de développement :

- Design Simple
- Conception Objet
- Technique de testing
- Clean Code
- Refactoring
- SOLID.

Acronymes, bonnes et mauvaises pratiques

SOLID :

- **S**ingle responsibility principle / Responsabilité unique :
  une classe, une fonction ou une méthode n'ont qu'un but bien défini

- **O**pen/closed principle / Ouvert/fermé
  une classe est ouverte à l'extension, mais fermée à la modification

- **L**iskov substitution principle / Substitution de Liskov
  une instance de type T doit pouvoir être remplacée par une instance de type G enfant de T, sans que cela ne modifie la cohérence du programme

- **I**nterface segregation principle / Ségrégation par interface
  Créer des interfaces spécifiques plutôt que générales

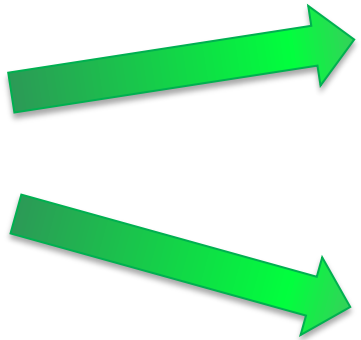
- **D**ependency inversion principle / Inversion des dépendances
  il faut dépendre des abstractions, pas des implémentations

Code SOLID en PHP

Principe de responsabilité unique :

Responsabilité multiple

```
interface UserModel {
    public function setId($id);
    public function getId();
    public function setName($name);
    public function getName();
    public function findById($id);
    public function insert();
    public function update();
    public function delete();
}
```



Responsabilité unique

```
interface UserModel {
    public function setId($id);
    public function getId();
    public function setName($name);
    public function getName();
}
```

Responsabilité unique

```
interface UserMapper {
    public function findById($id);
    public function insert(UserInterface $user);
    public function update(UserInterface $user);
    public function delete($id);
}
```

Dans le design pattern DAO, nous retrouvons deux classes qui ont chacun un but bien défini : UserModel contient les données tandis que UserMapper, les requêtes

Code SOLID en PHP

Ouverte à l'extension, mais fermé à la modification

Basé sur la délégation des responsabilités : Une classe parent délègue l'implémentation aux enfants

Fermé à l'extension

```
class Rectangle {
    public $largeur;
    public $hauteur;
}

class Tableau {
    public $rectangles = [];
    public function calculerAire() {
        $aire = 0;
        foreach ($this->rectangles as $rectangle) {
            $aire += $rectangle->largeur * $rectangle->hauteur;
        }
    }
}
```

Ce code est incorrect car si nous voulons utiliser un rond plutôt qu'un rectangle, nous devons modifier le Tableau

Code ouvert à l'extension :

Ouvert à l'extension

```php
interface Forme {
    public function aire();
}
class Rectangle implements Forme {
    public function aire() {
        return $this->largeur * $this->hauteur;
    }
}
class Cercle implements Forme {
    public function aire() {
        return $this->rayon * $this->rayon * pi();
    }
}
class Tableau {
    public function calculerAire() {
        $aire = 0;
        foreach ($this->formes as $forme) {
            $aire += $forme->aire();
        }
    }
}
```

Code SOLID en PHP

Substitution de Liskov :

Une clase enfant ne devrait pas réimplémenter le code métier de son parent. En suivant cette règle, l'enfant peut remplacer le parent où que ce soit.

Implémentation

```php
class Rectangle {
    public function setLargeur($w) { $this->largeur = $w; }
    public function setHauteur($h) { $this->hauteur = $h; }
    public function getAire() { return $this->hauteur * $this->largeur; }
}
class Carre extends Rectangle {
    public function setLargeur($w) { $this->largeur = $w; $this->hauteur = $w; }
    public function setHauteur($h) { $this->hauteur = $h; $this->largeur = $h; }
}
```

Utilisation

```php
function aireRectangle() {
    $rectangle = new Rectangle();
    $r->setLargeur(7); $r->setHauteur(3);
    $r->getAire(); // 21
}
```

Utilisation

```php
function aireRectangle() {
    $rectangle = new Carre();
    $r->setLargeur(7); $r->setHauteur(3);
    $r->getAire(); // 21
}
```

Code SOLID en PHP

Ségrégation par interface :

Les interfaces sont spécifiques pour éviter les implémentations inutiles

Interface lourde

```php
interface Travailleur {
    public function prendrePause();
    public function developper();
    public function appelerClient();
    public function faireReunion();
    public function etrePaye();
}
```

Implémentation

```php
class ChefProjet implements Travailleur {
    public function developper() { return false; }
}
class Developeur implements Travailleur {
    public function appelerClient() { echo ""; }
}
```

Certaines implémentations sont volontairement vide, ce qui rend le code lourd

Interfaces spécifiques

```php
interface Travailleur {
    public function prendrePause();
    public function etrePaye();
}
interface Coder {
    public function developper();
}
interface GestionClient {
    public function appelerClient();
    public function faireReunion();
}
```

Implémentation

```php
class Developeur implements Travailleur, Coder {}
class Manageur implements Travailleur, GestionClient {}
```

Abstraction : Utiliser une description plutôt que l'entité en elle-même

exemple : Si j'ai besoin de me déplacer, je ne parle pas de voiture mais de véhicule

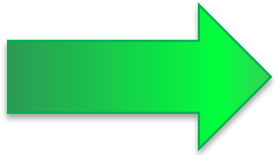
Interface : Souvent comparé à un contrat, dans le sens "règles de comportement"

KISS : Pour "Keep it simple, stupid". Indique qu'il ne faut pas chercher la complexité lorsqu'elle est inutile.

DRY : Pour "Don't Repeat Yourself". SI un code est redondant, il doit être extrait et factorisé

```
.message-erreur {
  border: 1px solid black;
  color: red;
  background: #fff;
  font-weight: bold;
}
```

```
.message-succes {
  border: 1px solid black;
  color: green;
  background: #fff;
  font-weight: bold;
}
```



```
.message {
  border: 1px solid black;
  background: #fff;
  font-weight: bold;
}
.message-erreur {color: red;}
.message-succes {color: green;}
```

YAGNI : Pour "you ain't gonna need it". Mettre en place le besoin et ne pas prévoir l'impossible.

Évite :

- La perte de temps, de la documentation et des tests supplémentaires
- D'ajouter des contraintes de fonctionnement
- L'alourdissement du code
- De créer du code qui ne sera peut-être pas utilisé

STUPID :

- **S**ingleton : Le design pattern Singleton (basé sur une instance unique) créé généralement un couplage fort et limite les évolutions
- **T**ight Coupling : Couplage fort, indique une dépendance forte vers une classe car on ne peut pas la retirer ni remplacer facilement
- **U**ntestability : Tests unitaires difficile, généralement dû aux couplages forts
- **P**remature Optimizations : L'optimisation prématurée rend le code difficilement lisible et n'est pas facilement maintenable
- **I**ndescriptive Naming : Nommage indéchiffrable dû aux abréviations et acronymes. Il vaut parfois mieux avoir un nom un peu plus long
- **D**uplication : Code redondant

Code Smell : Ensemble de mauvaises pratiques :

- **Code mort :** Code qui ne sera jamais utilisé.
- **Long Parameter List :** Trop de paramètres pour une fonction
- **Long Method :** Indique qu'une méthode/fonction est beaucoup trop longue et devrait être décomposée.
- **Large Class** : Classe ayant trop d'attributs
- **Feature Envy :** Méthode utilisant trop de méthodes d'une autre classe
- **Duplicated code** : code redondant