Bonnes pratiques pour les projets

Arnaud Labourel arnaud.labourel@univ-amu.fr

8 ou 15 avril 2019

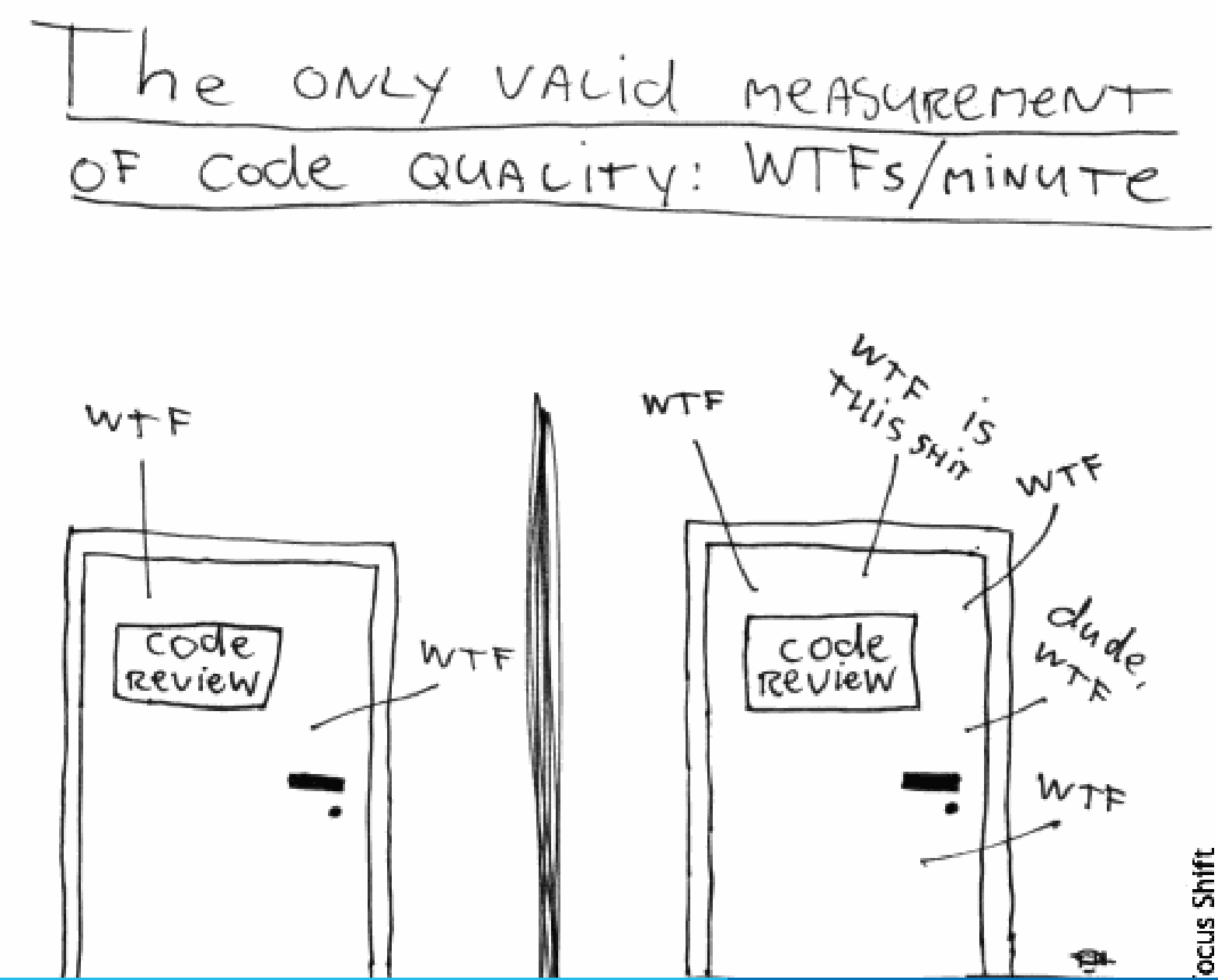
Bonnes pratiques de programmation

Coder proprement

Un programme propre :

- respecte les attentes des utilisateurs
- est fiable
- peut évoluer facilement/rapidement
- est compréhensible par tous (lisible par des humains)

Pourquoi coder proprement ?



Pourquoi coder proprement ?

Fonctionnel $\neq$ propre.

Pourquoi faire attention à son code :

- le rendre lisible :
  - pour soi-même
  - pour les autres
- le rendre facile à maintenir
- le rendre consistant

En résumé

On passe plus du temps à lire du code qu'à en écrire.

$\Longrightarrow$ Coder proprement, c'est gagner du temps !

Pourquoi programmer proprement ?

- Pour programmer les fonctionnalités les unes après les autres
- Pour ajouter des fonctionnalités à moindre coût
- Pour que les programmes soient utilisables plus longtemps
- Pour valoriser les codes écrits par les développeurs (car réutilisables)
- Pour effectuer des tests à toutes les étapes du développement
- Pour que les développeurs soient heureux de travailler

Une méthodologie pour bien nommer

Pourquoi bien nommer est important

Albert Camus (1944)

“Mal nommer un objet, c’est ajouter au malheur de ce monde”

Que veut dire le texte suivant ?

En ce qui concerne NSI, le DIU EIL qui a été présenté par l’IA IPR, lors de la réunion organisée par l’UFR, permettra de former les stagiaires, notamment dans le domaine de l’IA. Les IA des stagiaires vont d’ailleurs être réalisées sous peu.

Nommage

Important

- Les noms donnés aux variables/fonctions/fichiers sont essentiels pour la lisibilité du code.
- Un code bien écrit passe d'abord par des noms bien choisis.
- Des noms donnés au hasard et en dépit des conventions rendent le code illisible en cachant l'intention du code.

Exemple de mauvais nommage

Autre exemple de programme mal écrit :

```
class Rec:
    def __init__(self, L, l):
        self.L = L
        self.l = l

def get(R):
    C = []
    for r in R:
        if r.l == r.L:
            C.append(r)
    return C
```

Code corrigé

Programme légèrement refactoré :

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def is_square(self):
        return self.height == self.width

def find_squares(rectangles):
    squares = []
    for rectangle in rectangles:
        if rectangle.is_square():
            squares.append(rectangle)
    return squares
```

Pièges à éviter pour le nommage de variables/attributs

Vous ne devez pas avoir des variables avec des :

- noms en une lettre (même pour les indices) : `i`, `j`, ...
- noms numérotés : `a1`, `a2`, `a3`, ...
- abréviations ayant plusieurs interprétations : `rec`, `res`, ...
- noms ne donnant pas le sens précis : `temporary`, `result`, ...
- des noms trompeurs : par exemple un `account_list` doit être une `list` (et pas un `set` ou un autre type)
- types de l'objet au singulier pour une collection d'objet : une liste de personnes doit s'appeler `persons` et non `person`.
- noms imprononçables : `genymdhms`, ...

Règles de nommage

En respectant les conventions

La plupart des langages (Python inclus) ont des conventions pour l'écriture des noms et la manière de coder les espaces.
Pour python c'est la PEP 8 !

Manière d'écrire en Python

Utilisation du **snake case** (casse de serpent) dans la plupart des cas :
`words_separated_with_underscores`

Convention de nommage Python

- **snake_case** (commençant par une minuscule) : pour les méthodes, variables, attributs

  Exemples : `append`, `length`, `add_all`, ...

- **ALL_CAPS** : pour les constantes

  Exemples : `MAX_OVERFLOW`, `TOTAL`, ...

- **CamelCase** (commençant par une majuscule) : pour les noms de classes

  Exemples : `Rectangle`, `SmartCounter`, ...

Fonctions

Règles générales

- Des petites fonctions (une vingtaine de lignes max)
- Des `if`, `else`, `while` d'une ligne (qui appellent d'autres fonctions
- Un degré d'indentation raisonnable (pas plus de deux niveaux emboîtés)

Avec IDE

refactor → extract → method

pour extraire du code dans une fonction

Do One Thing

Une fonction ne doit faire qu'**une seule chose**.

Pour cela, elle ne doit réaliser que des étapes de même niveau d'abstraction.

On décompose la fonction :

Pour faire la cuisine je dois (premier niveau d'abstraction) :

- choisir une recette;
- réunir les ingrédients;
- suivre la recette.

Pour choisir une recette, je dois (deuxième niveau d'abstraction):

- réfléchir à ce que j'ai envie de manger;
- chercher sur marmiton.

Mauvaise approche

```
def cook():
    # On choisit la recette
    food_wanted = think_about_food()
    recipe = look_on_marmiton(food_wanted)

    # On réunit les ingrédients
    open_fridge();
    for ingredient in recipe.get_fresh_ingredients():
        take_in_fridge(ingredient);
    }
    closeFridge()
    openCupboard()
    ...
    # On suit la recette
    ...
```

Bonne approche

```
def cook():
    recipe = choose_recipe()
    gather_ingredients(recipe);
    follow_recipe(recipe);

def chooseRecipe():
    food_wanted = think_about_food()
    recipe = look_on_marmiton(food_wanted)
    return recipe

...
```

Nommer ses fonctions

Des noms :

- descriptifs;
- si besoin longs (mieux vaut long et compréhensible que court et mystérieux);
- cohérents.

Utilisez le refactor (Maj + F6 sur Pycharm par défaut) !

Les arguments

- Le moins d'arguments possibles (deux max dans l'idéal) :
  - en créant de nouvelles classes;
  - en faisant un méthode intermédiaire de la classe d'un des arguments.
- Pas d'arguments "flag";
- Des noms qui vont avec le nom de la méthode.

Don't Repeat Yourself

Éviter la duplication pour :

- gagner du temps à l'écriture;
- réduire le temps nécessaire à une modification;
- réduire le risque d'erreur.

Les commentaires

Éviter les commentaires

Besoin de commenter si le code n'est pas assez expressif.

$\Longrightarrow$ Rendre le code plus expressif.

Car il est **impossible** de maintenir des commentaires à jour.

Brian W. Kernighan et P.J. Plauger

Don't comment bad code. Rewrite it.

Bad code

```
# Check to see if the employee is eligible for full
if employee.age > 65
```

Good code

```
if employee.is_eligible_for_full_benefits():
```

Quelques bons commentaires

- explications d'intention (pourquoi faire de cette manière);
- une information sur les conséquences d'une méthode ("Ne lancez pas ça si vous n'avez pas le temps. . . ")
- un TODO;
- insister sur l'importance d'un passage;
- la documentation pour le code public !

Quelques mauvais commentaires

- celui qui répète ce que le code dit;
- le commentaire imprécis;
- le journal de modifications;
- le commentaire qui peut être remplacé par une fonction ou une variable;
- le commentaire gigantesque;
- et le pire : le code commenté.

La gestion de projet

Analyse fonctionnelle

1. Rechercher et caractériser les fonctions qu'un logiciel devrait avoir pour satisfaire les besoins de son utilisateur.
2. Hiérarchiser ces fonctions.

Utilisée pour **créer** mais aussi **améliorer** un logiciel (ou plus généralement un produit).

Identification des fonctions

- Fonction **principale** : la raison pour laquelle le logiciel est créé. Cette fonction peut être divisée en plusieurs fonctions simples.
- Fonction **contrainte** : conditions que le produit doit vérifier mais qui ne sont pas sa raison d'exister (par exemple la sécurité).
- Fonction **complémentaire** : ce qui facilite l'utilisation du logiciel, l'améliore ou le complète.

Cahier des charges

Un outil de l'analyse fonctionnelle : lister, décrire et hiérarchiser les fonctions.

Exemple de la tondeuse

Fonction principale :

- Couper l'herbe.

Fonctions contraintes :

- Respecter les normes de sécurité;
- Pouvoir être conservée à l'extérieur;
- S'adapter au terrain;
- Ne pas être trop bruyante;
- Ne pas être trop encombrante;
- etc,...

Fonctions complémentaires:

- Ramasser l'herbe;
- Être automatique.

Quelques notions de gestion de projet

- **Livrable** : produit destiné à la livraison : documentation, code, tests, etc. . .
- **Jalon** : fin d'une étape ou évènement important.

Le début d'un projet et sa fin sont des jalons. On fixe des jalons intermédiaires pour mesurer l'avancée du projet.

Un jalon peut être un livrable lié à une date.

La gestion de projet en informatique

Développer un logiciel implique de :

- comprendre le besoin du client;
- en tirer un cahier des charges;
- concevoir l'architecture du logiciel;
- développer le logiciel;
- tester s'il fonctionne comme prévu;
- le maintenir.

Deux types de management pour y parvenir :

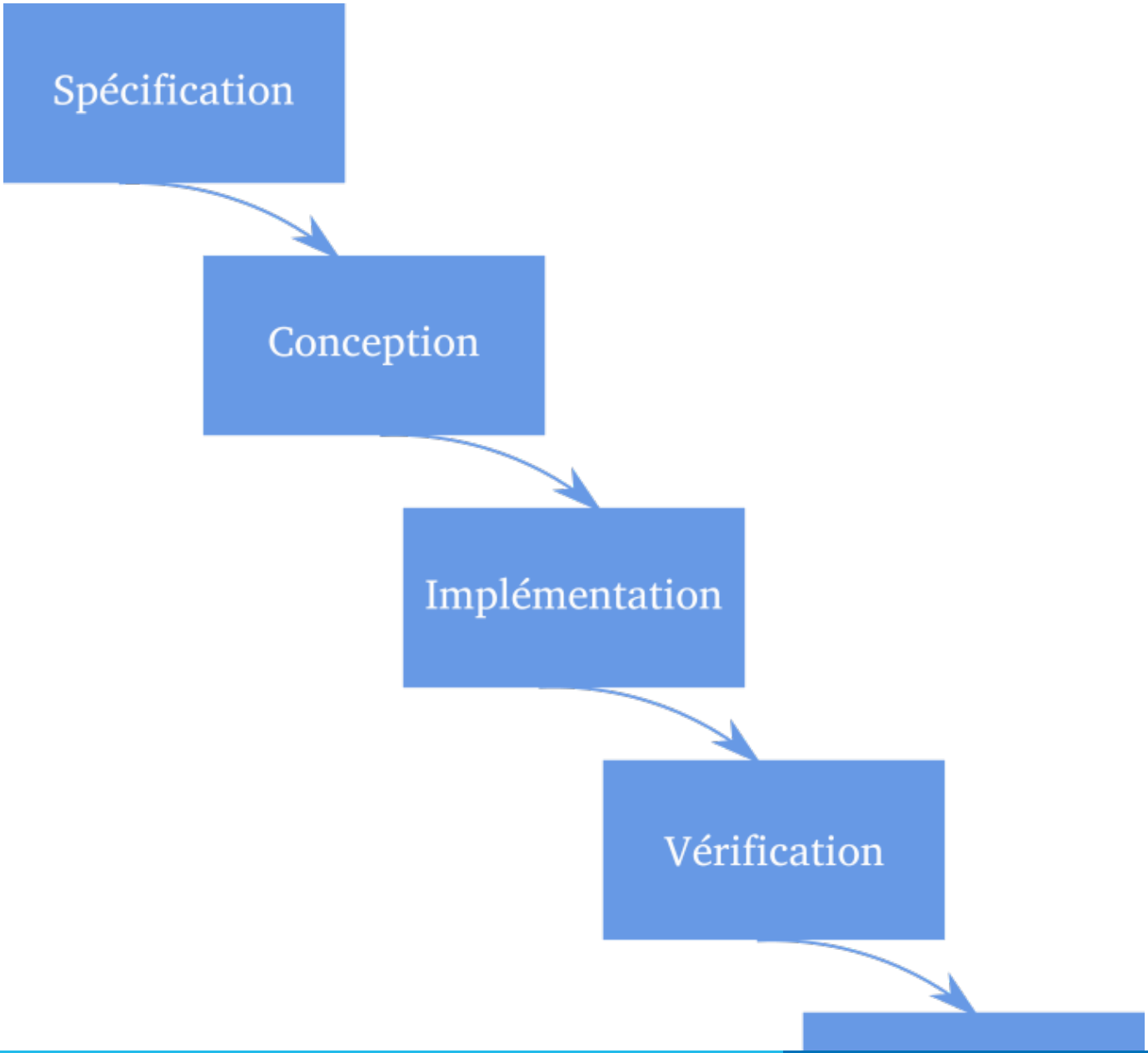
- méthodes traditionnelles;
- méthodes agiles.

Cascade (ou Waterfall)

Méthode traditionnelle, inspirée par le BTP.

Passage d'une phase à l'autre uniquement quand la précédente est terminée et vérifiée.

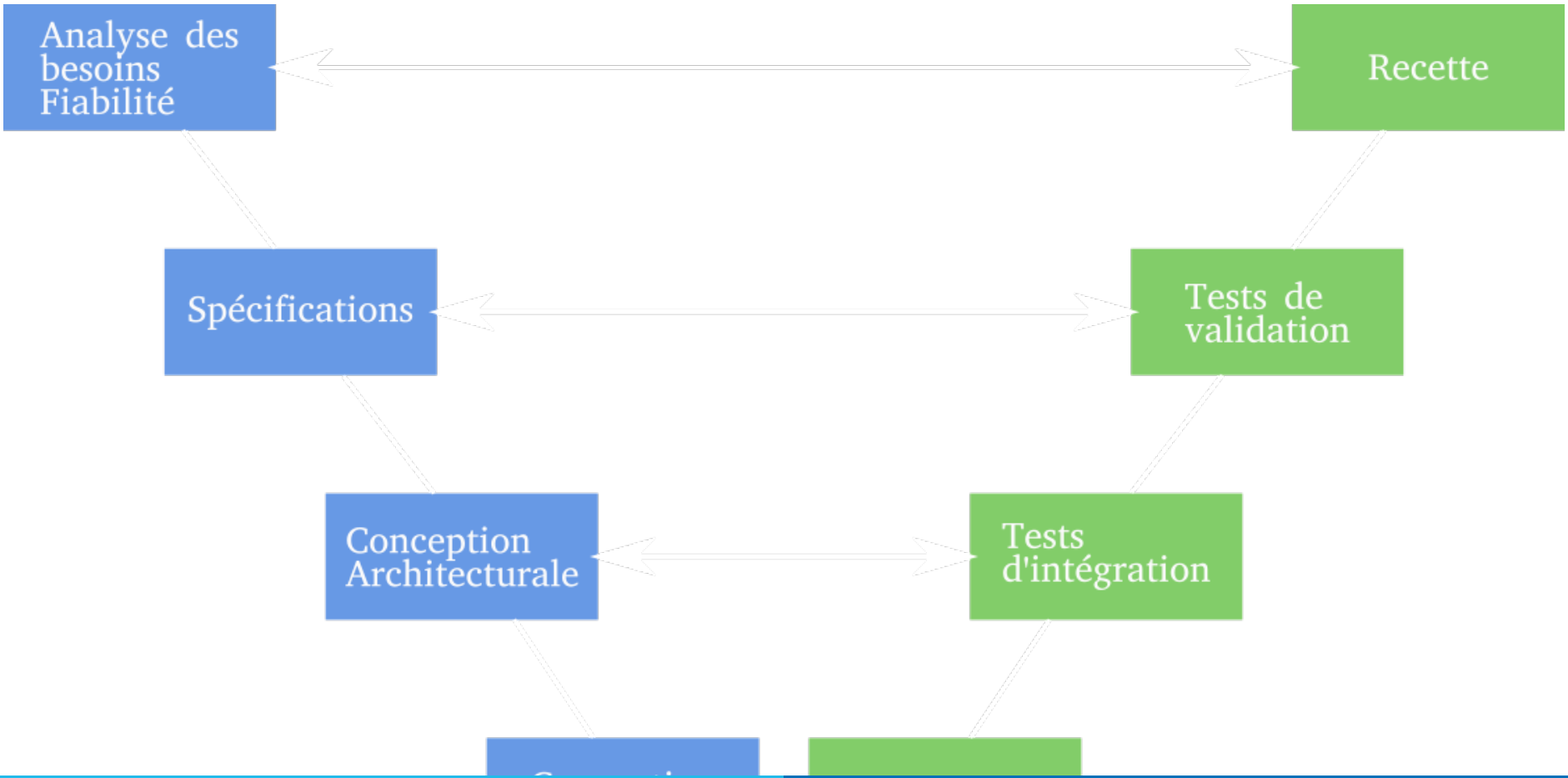
Cascade



Cycle en V

Un amélioration du modèle en cascade : limiter le retour aux étapes précédentes.

Standard depuis 1980.



Pro et con du cycle en V

Phase ascendante : renvoie de l'information sur la phase correspondante pour améliorer le logiciel.

Phase descendante : anticipation des attendus des étapes montantes.

Inconvénient : détache complètement la conception de la réalisation.

Agile

Objectifs :

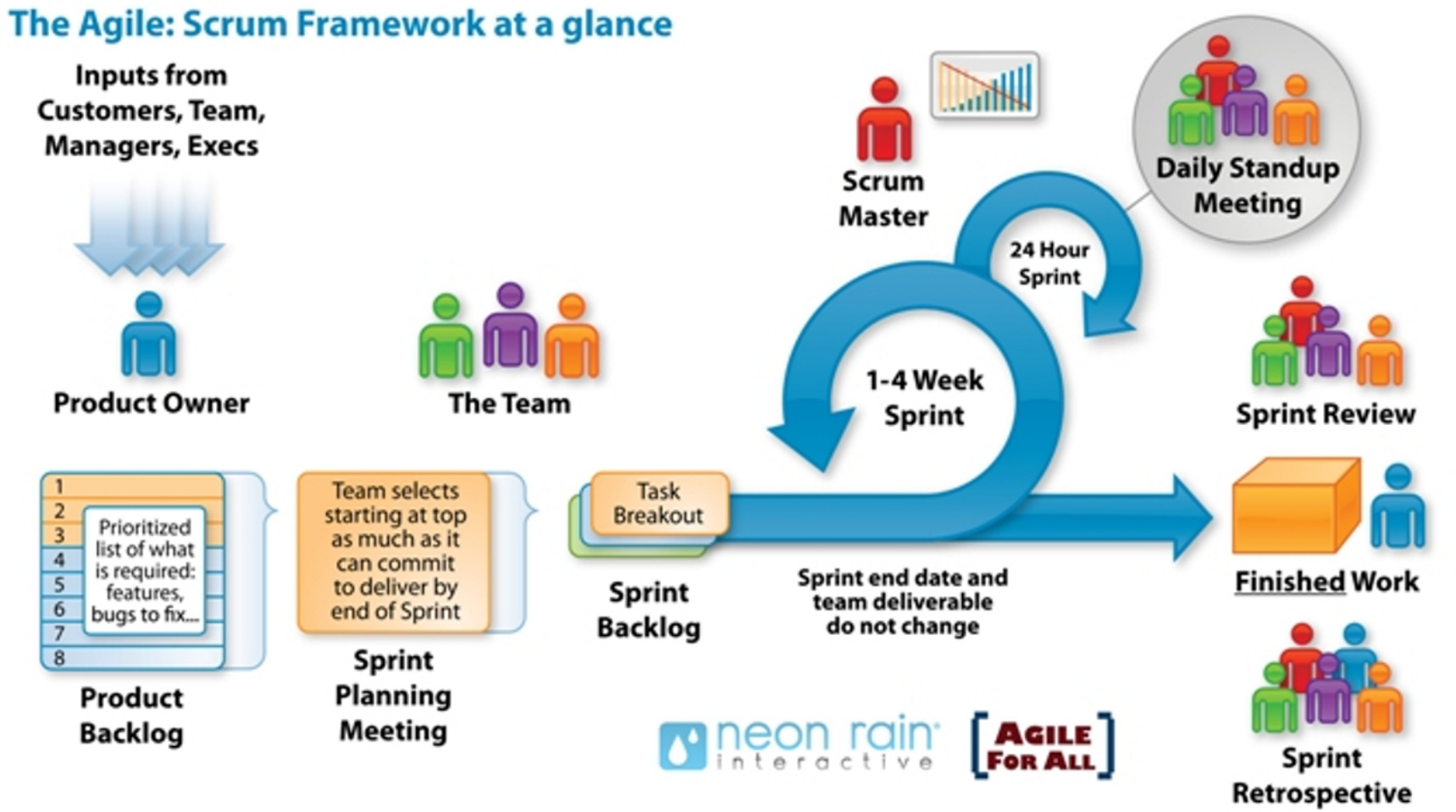
- plus pragmatique que les méthodes traditionnelles;
- impliquer le client pendant le développement;
- être réactif et s'adapter aux changements.

Définition par le manifeste Agile (2001).

Exemples :

- développement rapide d'applications (RAD);
- adaptative software development;
- extreme programming (XP);
- Scrum.

SCRUM



Source : agileforall.com

Gestion de version : Git

Principe général

Git est un **gestionnaire de versions** décentralisé.

Objectif : gérer l'évolution du contenu d'une arborescence de fichiers.

Intérêts :

- revenir à une version précédente du code;
- suivre les ajouts et modifications;
- travailler en parallèle sur différentes parties d'un projet;
- fusionner des modifications d'un même fichier par deux personnes différentes;
- ...

Fonctionnement

Un dépôt git est composé de trois zones distinctes :

- votre *répertoire de travail* : les fichiers sur votre disque dur;
- le *répertoire git* : l'arborescence de votre projet telle qu'elle était lors du dernier envoi;
- l'*index* ou *zone de staging* qui sert d'intermédiaire.

Étapes du développement :

1. Faire une modification du code dans le répertoire de travail.
2. Envoyer ces modifications dans l'index.
3. Envoyer ces modifications dans le répertoire git (on parle de **commit**).
4. Répéter 1. à 3.

Qu'est-ce qu'un bon commit ?

Un bon commit :

- ne concerne qu'une seule chose;
- est le plus petit possible;
- est cohérent;
- possède un message :
  - clair;
  - concis;
  - consistant avec le reste des commits;
  - qui décrit le "quoi" et pas le "comment".

Commandes les plus courantes

init et clone

Démarrer un dépôt git :

- depuis un dépôt existant sur un autre serveur :

```
git clone git@github.com:user/project.git
```

Depuis Github : bouton “Clone or download” pour cloner le repo via SSH ou HTTPS.

- dans un répertoire existant dans votre espace de travail :

```
git init
```

Attention : initialisation du dépôt mais aucun fichier versionné pour l'instant.

add

Passer des fichiers de l'espace de travail à l'index :

```
git add [files]
```

Par exemple, pour ajouter tous les fichiers `.py` :

```
git add *.py
```

commit

Passer des fichiers de l'index au dépôt :

```
git commit
```

En ajoutant un message de commit :

```
git commit -m "Message de validation"
```

Pour passer d'un coup du répertoire de travail au dépôt en ajoutant **tout** (attention à bien être sûr qu'on veut tout inclure dans le commit):

```
git commit -am "Message de validation"
```

status et log

Connaître les changements à valider :

```
git status
```

Afficher l'historique des commit de la branche :

```
git log
```

Les branches

Principe

Permettent de développer en parallèle.

Source : leanpub.com

Pour :

branch et checkout

Créer une branche :

```
git branch nomDeLaBranche
```

Se déplacer sur une branche :

```
git checkout nomDeLaBranche
```

Faire les deux d'un coup :

```
git checkout -b nomDeLaBranche
```

Voir les branches et sur laquelle on se trouve :

```
git branch
```

Interaction avec le dépôt distant

pull

Pour récupérer sur la branche local sur laquelle vous vous trouvez, les modifications de la branches distante correspondante :

```
git pull
```

Attention la branche distante doit être dans un état descendant directement de votre dépôt local.

push

Pour envoyer les commit locaux de la branche courante sur la branche distante du même nom :

```
git push
```

Pour spécifier quoi pousser où :

```
git push nomDistant nomDeBranche
```

Lors du premier push, pour lier la branche distante et locale :

```
git push -u origin nomDeBranche
```

Tests unitaires

Pourquoi les tests unitaires ?

- assurer la correction du code;
- trouver rapidement les bugs pendant le développement;
- identifier des manques dans les spécifications;
- faciliter les modifications.

Le développement par les tests (TDD)

Cycle de développement du TDD

1. Écrire un test : définit une fonction ou l'amélioration d'une fonction.
2. Lancer le test : il doit échouer (pour montrer que le test fait référence à une fonctionnalité qui n'existe pas encore, et qu'il fonctionne bien).
3. Écrire le code qui fait passer le test (et rien d'autre).
4. Lancer les tests : si le test ne passe pas, retourner à l'étape 3.
5. Refactorer : déplacer du code si besoin, supprimer la duplication, vérifier les noms...

Théorisé par Kent Beck

Intérêts

- réfléchir à ce que fait le code avant de coder;
- garantir que tout est testé (puisque rien n'est écrit sans test);
- réduire significativement le temps passé à debugger;
- avancer par petits pas;
- voir son avancée.