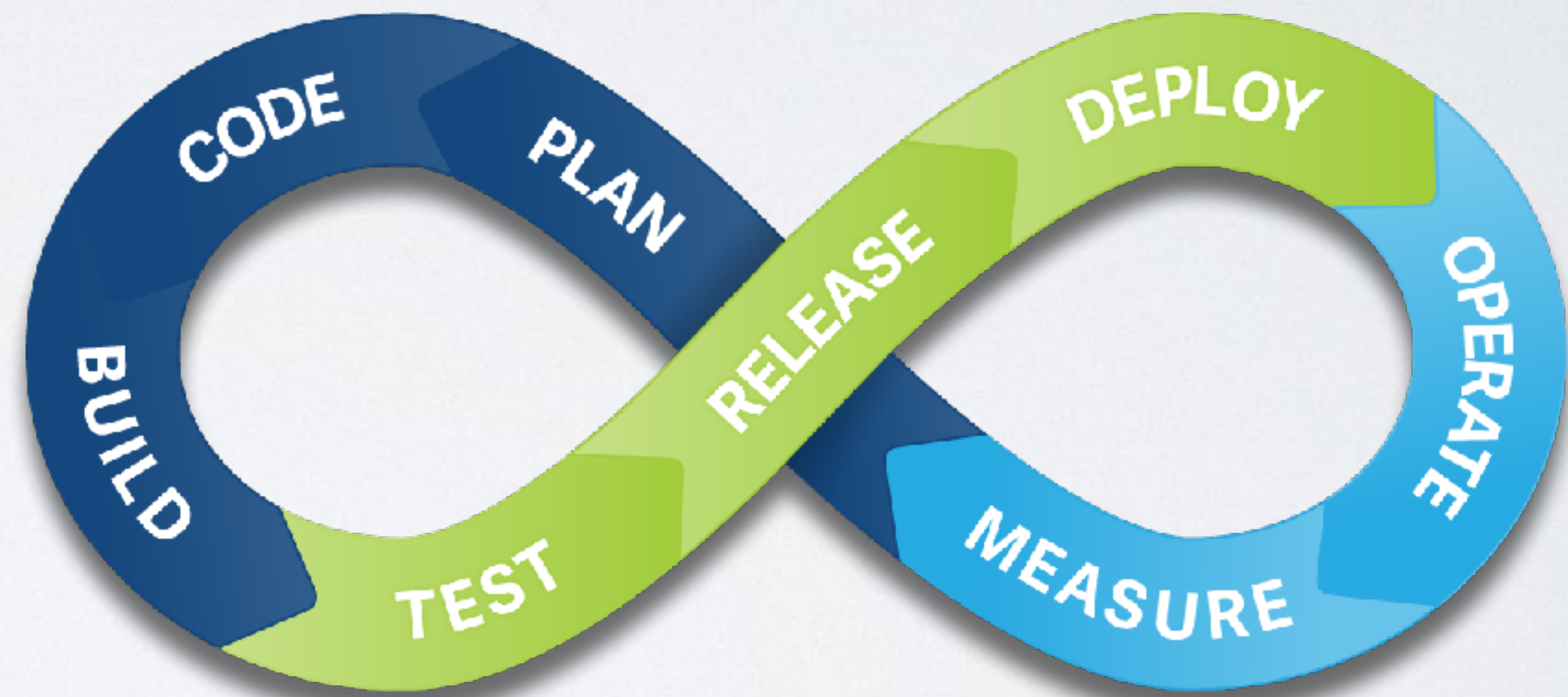


# IPI2: PROJET COLOR FLOOD



<https://play.google.com/store/apps/details?id=com.wetpalm.colorflood&hl=fr>



# PROJET COLORFLOOD

## I. Programmer le jeu

- Une grille, des couleurs
- Choisir une couleur
- Nombre de coups limité







# PROJET COLORFLOOD

## 2. Programmer un solveur

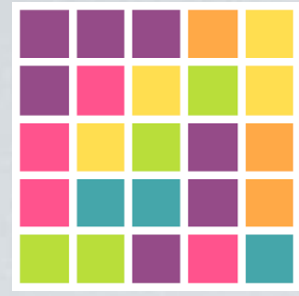
- Par essais successifs : force brute

## 3. Programmer un jeu complet

- Génération d'une grille aléatoire
- Calcul du nombre de coups autorisé







# PROJET COLORFLOOD

## Solveur en force brute

```
solveur(grille g, entier n, pile solution) {  
    // déclarations des variables locales i, g2, ...  
    for (i=0; i<6; i=i+1) { // pour toutes les couleurs possibles  
        solution = empiler(solution, c[i]);  
        g2 = propageCouleur(g, c[i]);  
        if (terminaison(g2)) uneSolutionTrouvee(solution);  
        else solveur(g2, n, solution);  
        depiler(&solution);  
    }  
}
```





# DÉROULEMENT

- Organisation du travail
  - Des équipes
  - Des étapes
  - Des livrables
  - Des outils



# LES ETAPES





# LES ETAPES

## **Etape I**

- Structures de données : la grille(.h)
- Fonctions de manipulation (.c)
  - allouer, initialiser (aléatoire, fichiers), afficher, libérer, terminaison, ...
- Documentation
- Tests unitaires



# LES ETAPES

## **Etape 2** : programmation d'un Color Flood jouable

- choix de la dimension
- boucle de jeu
- exécution possible en mode terminal
- alternatives : SDL, Qt, ...





# LES ETAPES

**Etape 3** : programmation du solveur



# LES ETAPES

**Etape 4** : programmation du jeu complet avec solveur intégré.

- Heuristique.





# NOTION D'HEURISTIQUE

Quand on joue on ne choisit pas au hasard la couleur à jouer mais on applique une stratégie.

Caractérisation de cette stratégie : **heuristique**

Parcours arbre des solutions : « meilleur » choix d'abord

- Arbre « physique » des solutions
- Initialement : 1 noeud = grille initiale

Boucle :

- Recherche du « meilleur » noeud potentiel
- Détermination de la meilleure couleur à jouer ;
- Propagation de la couleur et test de solution atteinte ;
- Ajout de la nouvelle configuration à l'arbre.



# GESTION DU PROJET



# LA MÉLÉE (SCRUM)

## RUGBY A XV

### LE GROUPE DES AVANTS



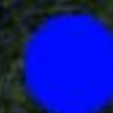
1 ÈRE LIGNE  
NUMÉRO 1 ET 3 : PILIERS  
NUMÉRO 2 : TALONNEUR



2 ÈME LIGNE



3 ÈME LIGNE



DEMI DE MÉLÉE

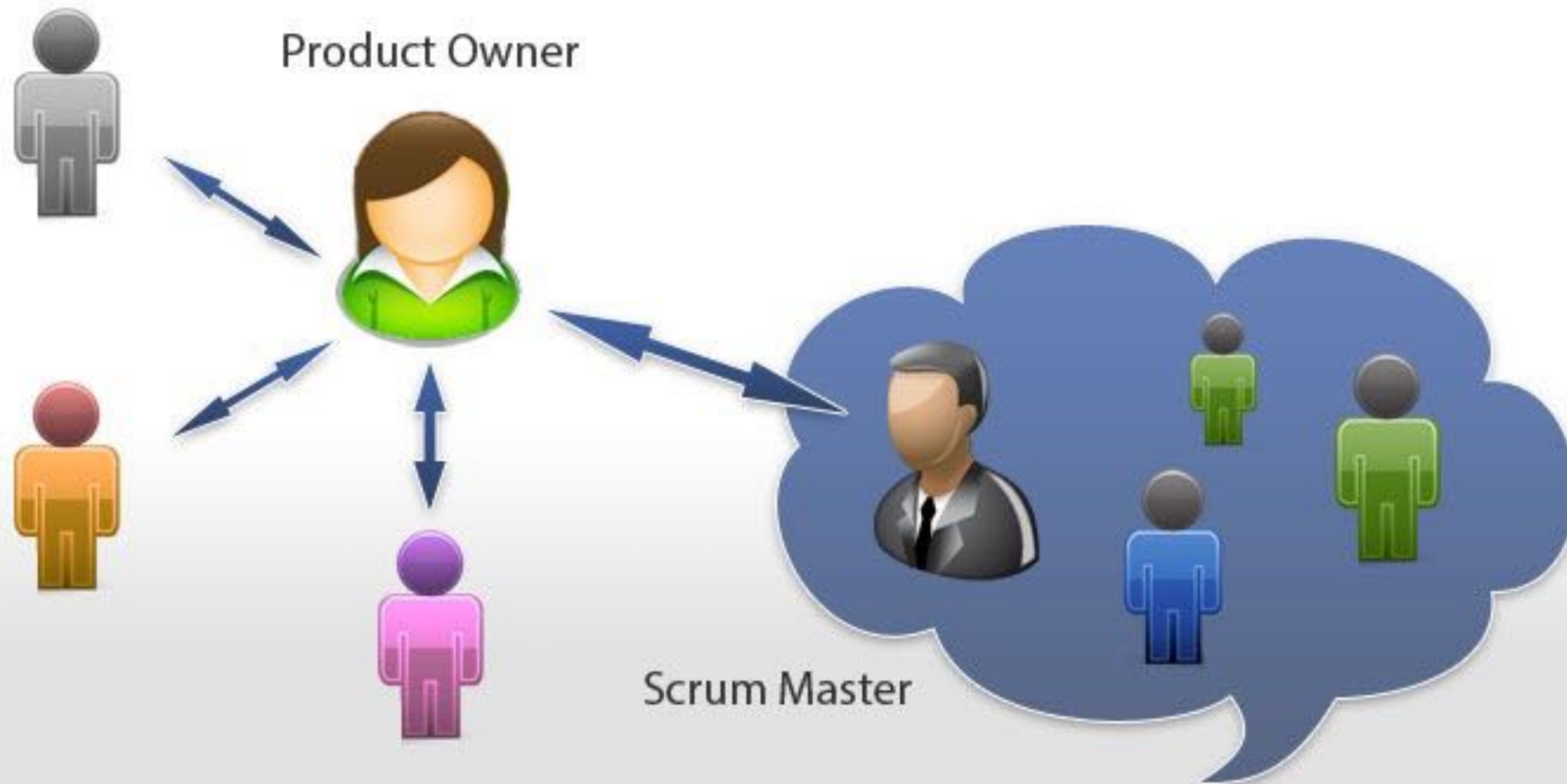


# TUNNEL





# SCRUM TEAM







# SCRUM TEAM

- Scrum Master: leader, animateur, protecteur, assure l'agilité de l'équipe.
- Development team: délivre et assure la qualité du produit.
- Manager: créé la structure et assure la stabilité.
- Client: demande le produit, peut-être responsable du budget.
- Product Owner: dirige le projet d'un point de vue métier, vision claire du produit.
- User: expertise du domaine adressé, source privilégiée pour fixer les priorités.





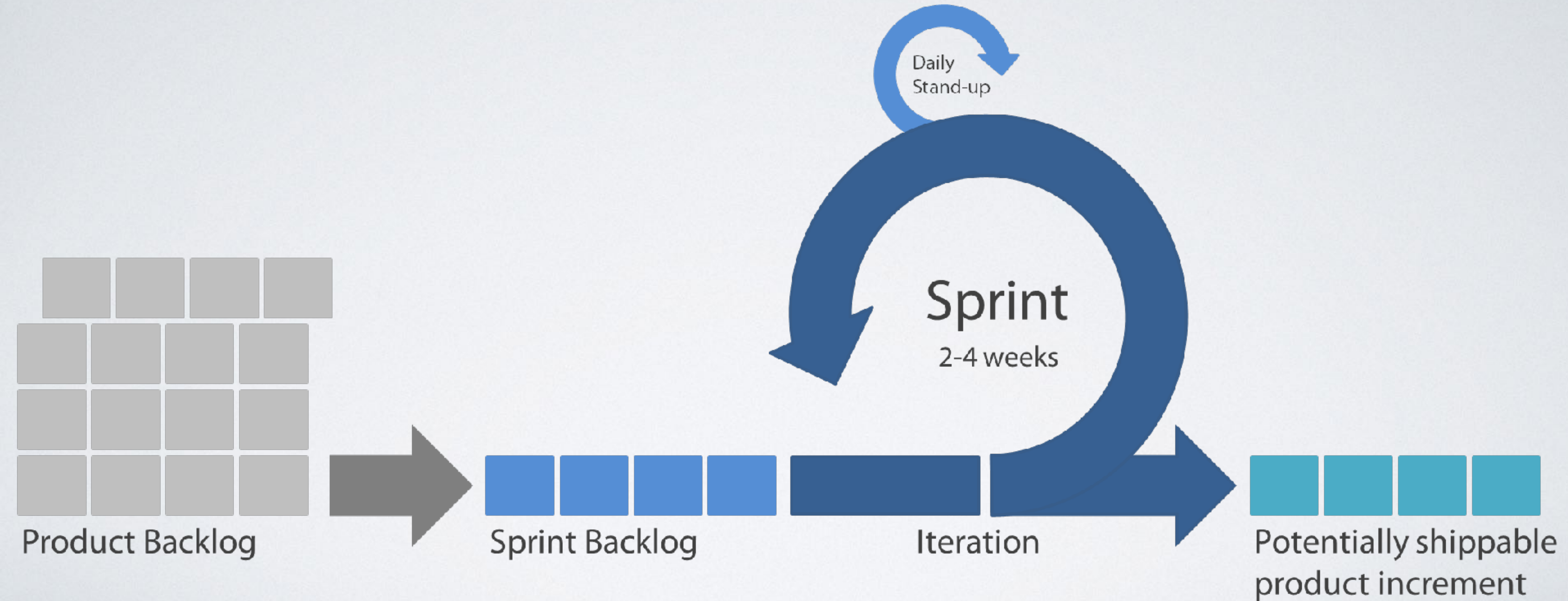
# SCRUM TEAM

- Pas de hiérarchie au sein de l'équipe
- Décisions collégiales
- Répartition des tâches collégiale
- Tous les membres participent à la conception
- La communication et l'esprit d'équipe sont favorisés





# CYCLES SCRUM





# CYCLES SCRUM

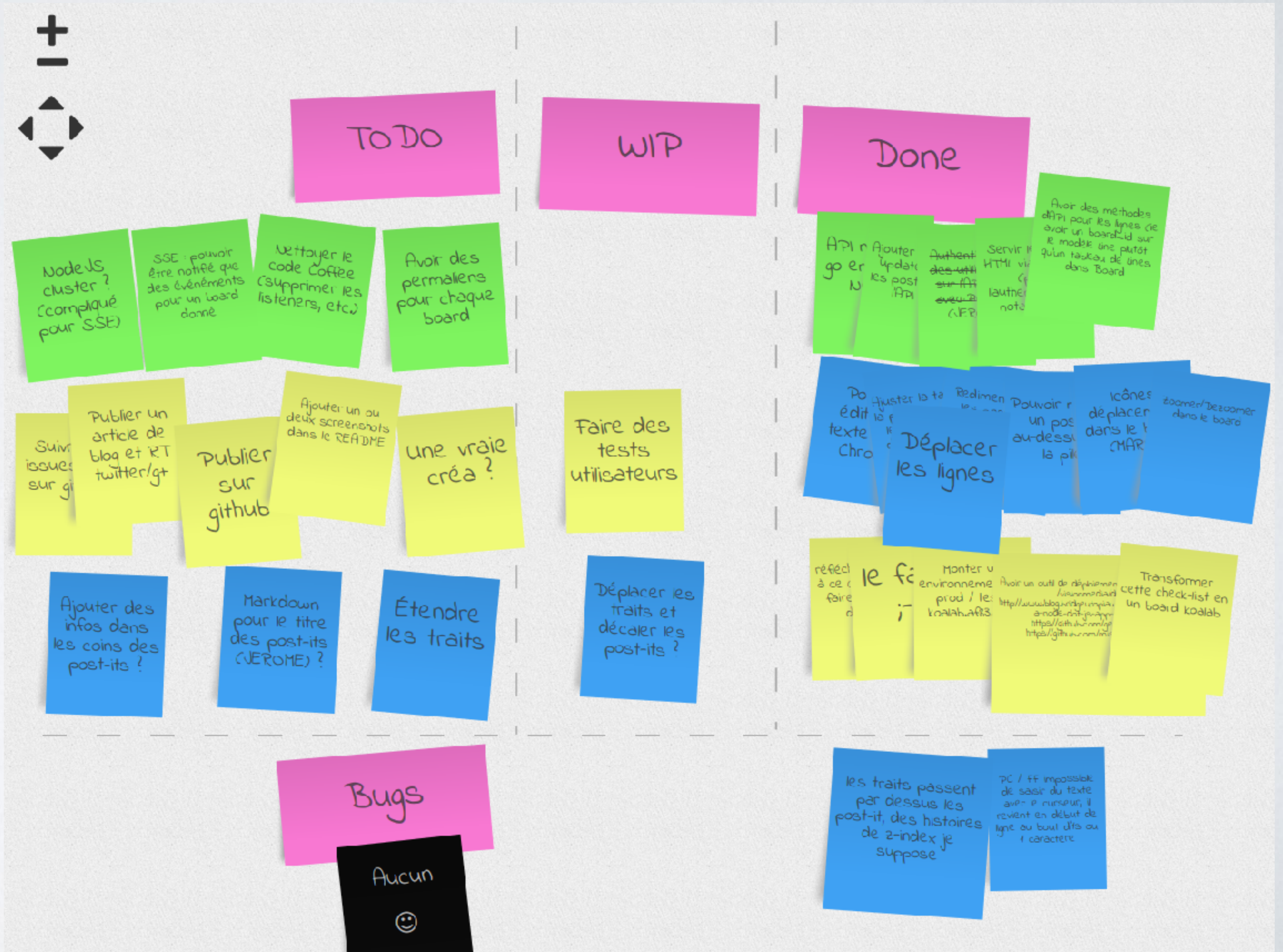
- **Product Backlog:** Ensemble des fonctionnalités du produit, avec priorités
- **Sprint Backlog:** Ensemble des fonctionnalités élues pour le Sprint
- **Sprint:** Intégration de l'itération





# CYCLES SCRUM

- **Tableau des tâches**



- **Suivi collaboratif**





# GESTION DU PROJET

- Organisation : **Collaborative**
  - Equipes de 4 personnes
  - Définition et répartition des rôles sous la responsabilité des étudiants
  - 1 référent dans chaque équipe, qui est en capacité de présenter à n'importe quel moment :
    - L'organisation de l'équipe, les rôles de chacun
    - Le suivi d'avancement
  - Le référent change à chaque sprint (1 sprint = 1 étape = 1 lot)
- Méthode : **Agile**
- Suivi de l'avancement : **Trello**





# COLLABORATION:TRELLO

- Chaque équipe crée un **tableau** (selon structure bien définie)
- Les étudiants sont « **membres de l'équipe** » dans Trello
- Les coach sont « **membres en dehors de l'équipe** »

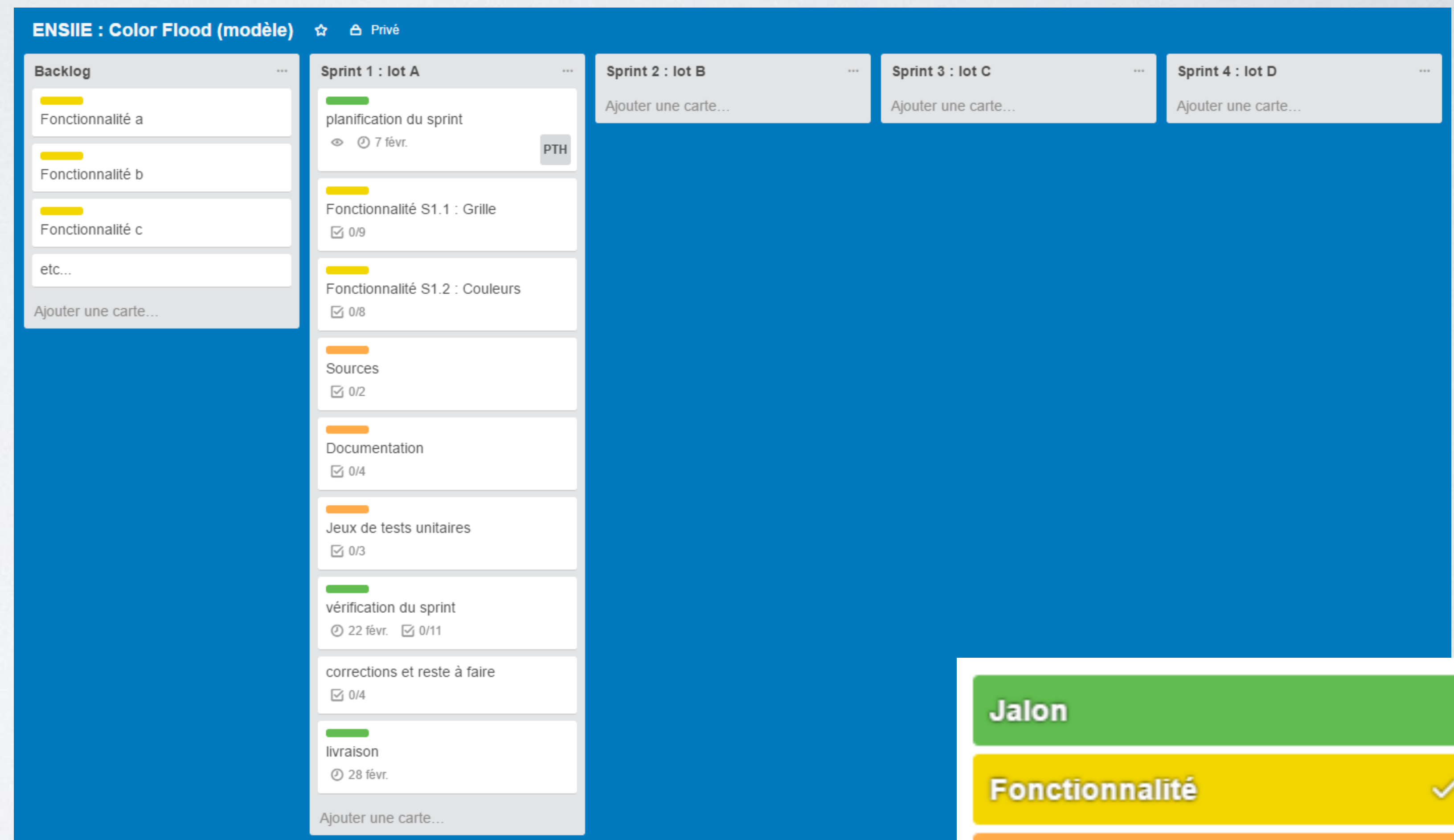
Ceci **vous** permettra et **nous** permettra de suivre votre avancement



# TABLEAU DU PROJET

## Permet de gérer

- Le backlog des fonctionnalités
- La composition des Sprint
- Le planning
- L'affectation des tâches
- L'avancement







# CYCLE DE VIE

Vous réalisez un logiciel qui répond à des **fonctionnalités**.

Vous fabriquez les **livrables** en respectant les **jalons**.

Pour chaque sprint :

- 3 jalons :
  - Planification du sprint : au plus tard pour les séances de TD
  - Vérification du sprint : 1 semaine avant la livraison
  - Livraison : dates imposées

Jalon

Fonctionnalité ✓

Livrable





# CYCLE DE VIE

- 3 livrables :
  - Sources (code, makefile)
  - Documentation (rapport, Readme)
  - Jeux de tests unitaires
- N fonctionnalités : à vous de les déduire du cahier des charges


Jalon	
Fonctionnalité	✓
Livrable	





# TABLEAU DU PROJET

- Composées de 2 checklist
  - Exigences
  - Tâches
- Selon la nature de la carte, les exigences peuvent être imposées
- Les tâches sont toujours libres : à vous de les définir

 **Fonctionnalité S1.1 : Grille**  
Dans la liste [Sprint 1 : lot A](#)  
Étiquettes  
**Fonctionnalité** +  
[Modifier la description...](#)

☒ **Exigences** [Supprimer...](#)  
0%  
☐ Allocation d'une grille carrée de couleurs de taille variable  
☐ Libération de l'espace mémoire occupé par une grille  
☐ Initialisation de la grille à partir de valeurs aléatoires  
☐ Initialisation de la grille à partir de valeurs contenues dans un fichier (pour la répétition des tests).  
[Ajouter un élément...](#)

☒ **Tâches** [Supprimer...](#)  
0%  
☐ Tâche 1  
☐ Tâche 2  
☐ Ecrire une fonction qui permet de créer des fichiers  
☐ Tâche 4  
☐ etc...  
[Ajouter un élément...](#)





# TABLEAU DU PROJET

- Convention terminologique
  - **Exigence** : un résultat à obtenir (verbe proscrit)
  - **Tâche** : toujours un verbe d'action





# LISTE DES EXIGENCES

Nature de la carte	Carte	Exigences
Cartes « jalon »	Planification du sprint	<ul style="list-style-type: none"><li><input type="checkbox"/> Référent désigné</li><li><input type="checkbox"/> Fonctionnalités identifiées</li><li><input type="checkbox"/> Rôles définis / tâches réparties</li><li><input type="checkbox"/> Planning défini</li></ul>
	Vérification du sprint	<ul style="list-style-type: none"><li><input type="checkbox"/> Développement en C</li><li><input type="checkbox"/> Codes sources gérés avec un dépôt Git</li><li><input type="checkbox"/> Codes compilés avec les options -Wall -Wextra</li><li><input type="checkbox"/> La compilation ne produit aucun warning</li><li><input type="checkbox"/> Makefile fourni pour le lot</li><li><input type="checkbox"/> Absence de fuites de mémoire vérifiée avec Valgrind</li><li><input type="checkbox"/> Commentaires au format Doxygen</li><li><input type="checkbox"/> Documentation générable à la demande grâce à une cible dans le Makefile</li></ul>
	Livraison	<ul style="list-style-type: none"><li><input type="checkbox"/> Livrables fournis à Pierre Tellier</li></ul>





# LISTE DES EXIGENCES

Nature de la carte	Carte	Exigences
Cartes « livrables »	Sources	<input type="checkbox"/> Codes source <input type="checkbox"/> Fichier Makefile
	Documentation	<input type="checkbox"/> Rapport : Algorithmes <input type="checkbox"/> Rapport : Organisation adoptée (répartition du travail, planning) <input type="checkbox"/> README : Instructions d'installation <input type="checkbox"/> README : Instructions d'utilisation
	Jeux de tests unitaires	<input type="checkbox"/> (à vous de les définir en fonction du lot)
Cartes « fonctionnalités »	à vous de les déduire du cahier des charges	





# ON SE LANCE!

- Organisez-vous : désignation du référent, des rôles de chacun
- Décrivez toutes les fonctionnalités dans le backlog (lots A à D)





# ON SE LANCE!

- En début du Sprint :
  - Préparez le sprint (selon le modèle) : cartes de **Jalons** et **Livrables**
  - Ajoutez les cartes de **Fonctionnalités** à développer pour le sprint
  - Complétez pour chaque carte :
    - Les exigences
    - Les tâches
    - La date d'échéance
    - Les acteurs (membres)



# OUTILS





# OUTILS

- Documentation: **Doxygen**
- Système de gestion de version: **Git**
- Outil d'analyse: **Valgrind**
- Tests unitaires: **CUnit**





# DOCUMENTATION

- **Doxygen** permet une génération automatique d'une documentation
- Se base sur les commentaires du code
- Code bien commenté = code lisible + documentation





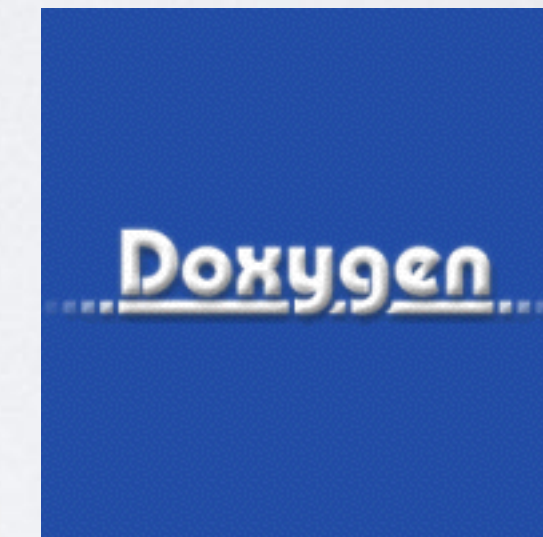


# DOCUMENTATION

```
/**  
  \file [<name>]  
  \brief {brief description}  
  \author { list of authors }  
**/
```

```
/**  
  \struct  
**/
```

```
/**  
  \fn  
  \param  
  \return  
**/
```





## Str\_t Struct Reference

Objet chaîne de caractères. [More...](#)

### Data Fields

char *	sz
size_t	len

### Detailed Description

Objet chaîne de caractères.

**Str\_t** est un petit objet de gestion de chaînes de caractères. La chaîne se termine obligatoirement par un zéro de fin et l'objet connait la taille de chaîne contient !

Definition at line **24** of file **test3.c**.

### Field Documentation

◆ len

size\_t len

Taille de la chaîne sz sans compter le zéro de fin.

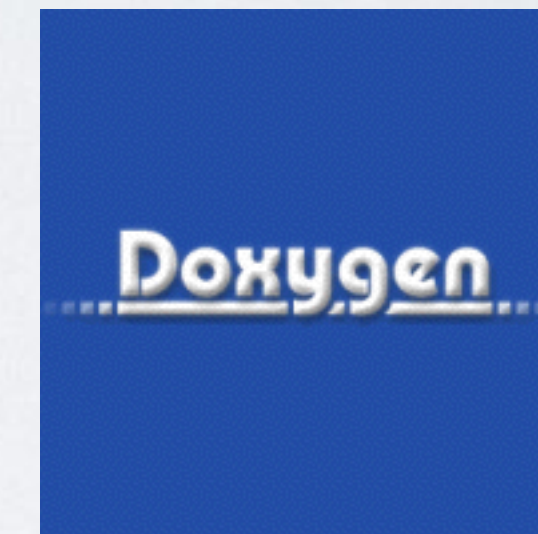
Definition at line **27** of file **test3.c**.





# DOCUMENTATION

- <http://www.stack.nl/~dimitri/doxygen/manual>







# GESTION DE SOURCES

- **Git** indexe des fichiers décentralisés en se basant sur leur somme de contrôle
  - `git init`: créé un nouveau dépôt
  - `git clone`: clone un dépôt distant
  - `git add`: ajoute de nouveaux objets (fichiers) dans la base
  - `git commit`: soumet un nouveau changement à la base
  - `git branch`: créer une nouvelle branche de développement
  - `git merge`: fusionne deux branches
  - ...







# GESTION DE SOURCES

## • **GitHub**

- Service Web d'hébergement et de gestion de source
- Basé sur l'outil Git
- Idéal pour le travail collaboratif
- Centralise l'information







# GESTION DE SOURCES

- <https://git-scm.com/documentation/>



- <https://help.github.com/>



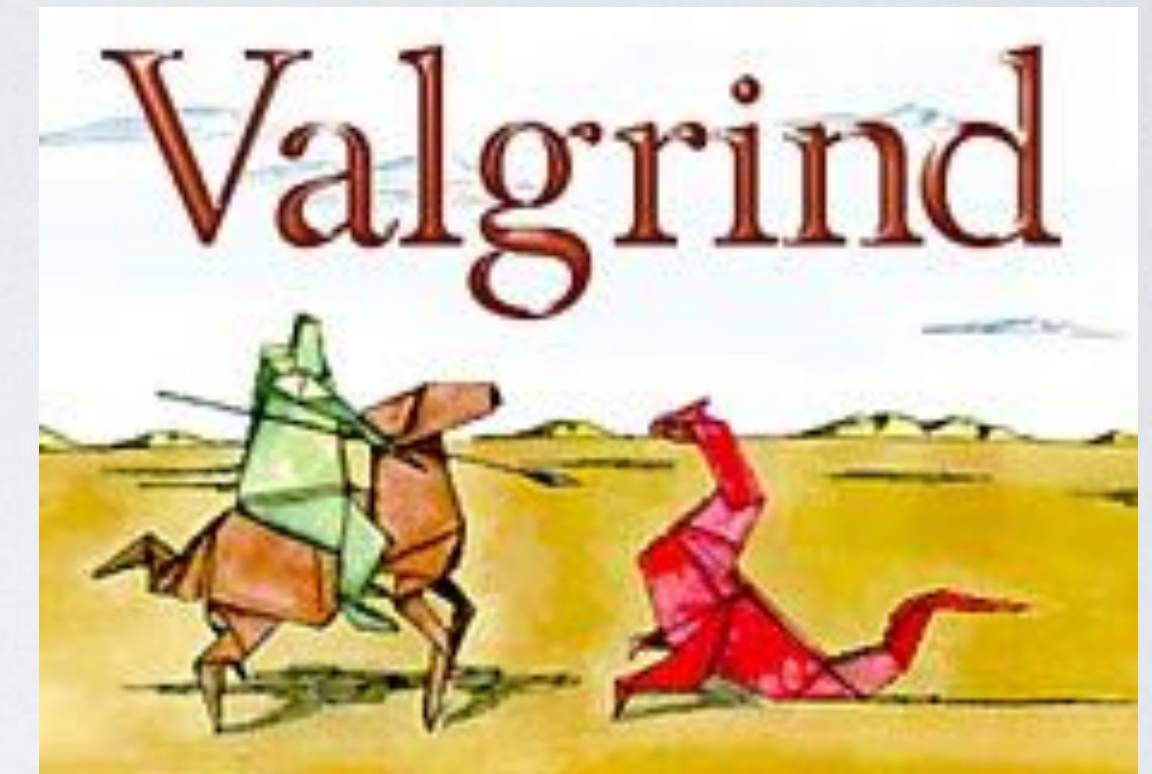




# PROFILEUR

- **Valgrind**

- Profileur
- Spécialisé dans la détection de fuites mémoire
- Utilisation simple:



```
gcc colorflood.c -o colorflood -g -O0
```

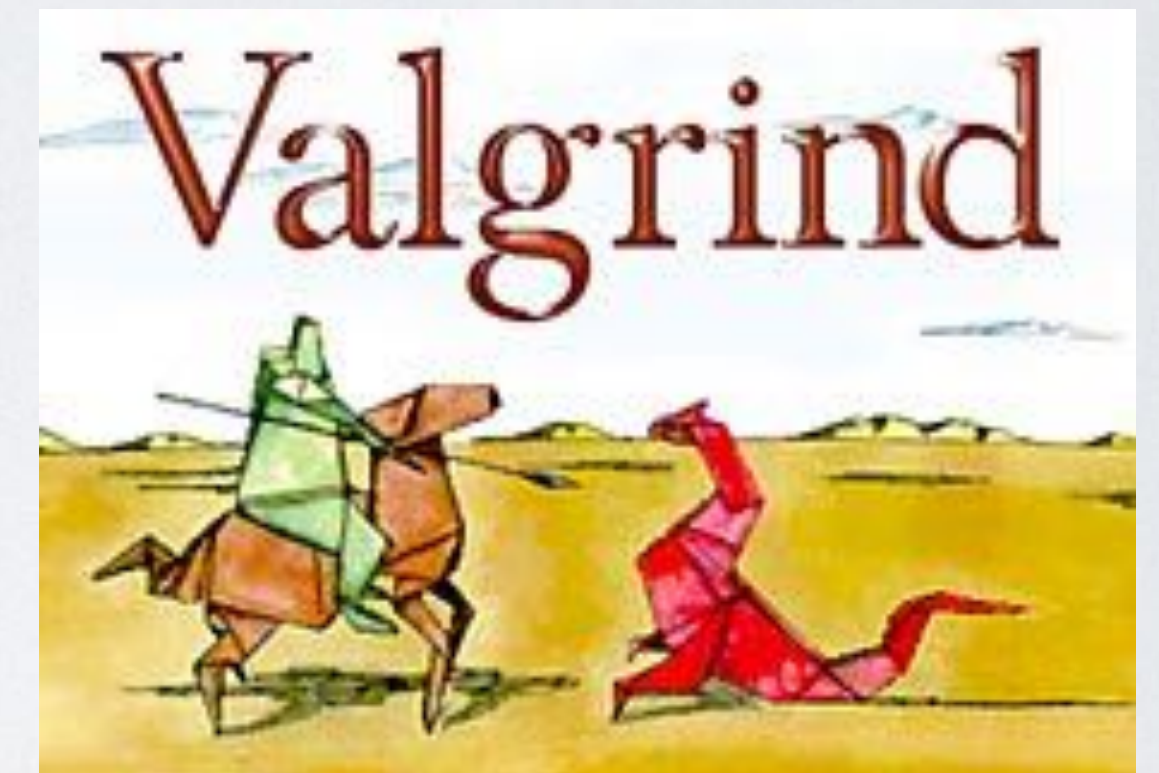
```
valgrind --leak-check=yes colorflood arg1 arg
```





# PROFILEUR

<http://valgrind.org/docs/manual/index.html>







# TESTS UNITAIRES

- **CUnit** est un framework permettant l'écriture et l'exécution de tests unitaires
  - Écriture simple de jeux de tests pour un ensemble de fonctions
  - Exécution de tests simplifiée
  - Facilite l'automatisation de ces tests
  - Idéal pour effectuer du **TDD**: Test Driven Development





# TESTS UNITAIRES

- Exemple sans framework:

```
int valeurCible(grille g, int n) {  
    int res, minG=99, maxG=0;  
    // . . .  
    return res;  
}  
  
int test_valeurCible(void) {  
    int g1[]={8,12,17,16,13,1,4,14,3};  
    assert(valeurCible(g1,3) == 9);  
    int g2[]={8,1,4,3};  
    assert(valeurCible(g2,2) == 4);  
    // . . .  
    fprintf(stderr, "test 1 valeurCible OK\n");  
}
```

- Problèmes

- Arrêt au 1er échec ...
- Nécessite de documenter l'exécution





# TESTS UNITAIRES

- Avec le framework **CUnit**

```
int test_valeurCible(void) {  
    int g1[]={8,12,17,16,13,1,4,14,3};  
    CU_assert(valeurCible(g1,3) == 9);  
    int g2[]={8,1,4,3};  
    CU_assert(valeurCible(g2,2) == 4);  
    // . . .  
}
```

```
int main() {  
    CU_pSuite pSuite = NULL;  
    CU_initialize_registry();  
    pSuite = CU_add_suite("Suite",NULL,NULL);  
    CU_add_test(pSuite,"test de valeurCible()",test_valeurCible);  
    CU_basic_set_mode(CU_BRM_VERBOSE);  
    CU_basic_run_tests();  
    CU_cleanup_registry();  
    return 0;  
}
```

- Un registre de tests
- Des suites
- Des tests





# TESTS UNITAIRES

<http://cunit.sourceforge.net/documentation.html>



# EXTREME PROGRAMMING





# EXTREME PROGRAMMING

- **Principes**

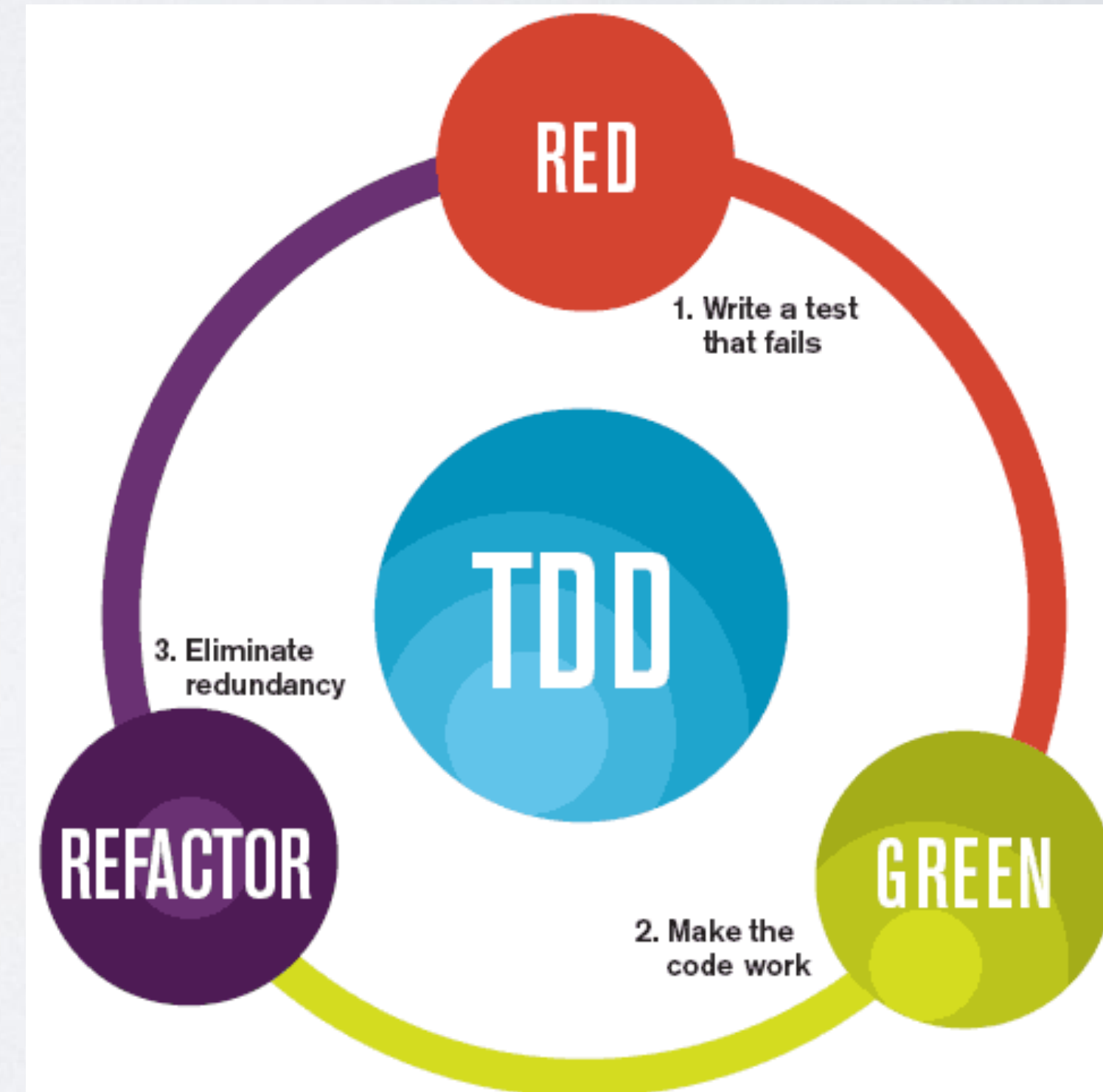
- Méthodes agiles appliquées au développement logiciel
- Feedback rapide et constant
- Processus fluide et continue





# EXTREME PROGRAMMING

- Pratiques: Tests
  - TDD: Tests Driven Development
  - On écrit d'abord les tests, puis les fonctionnalités
  - Les programmeurs s'occupent des tests unitaires
  - Le client s'occupe des tests fonctionnels

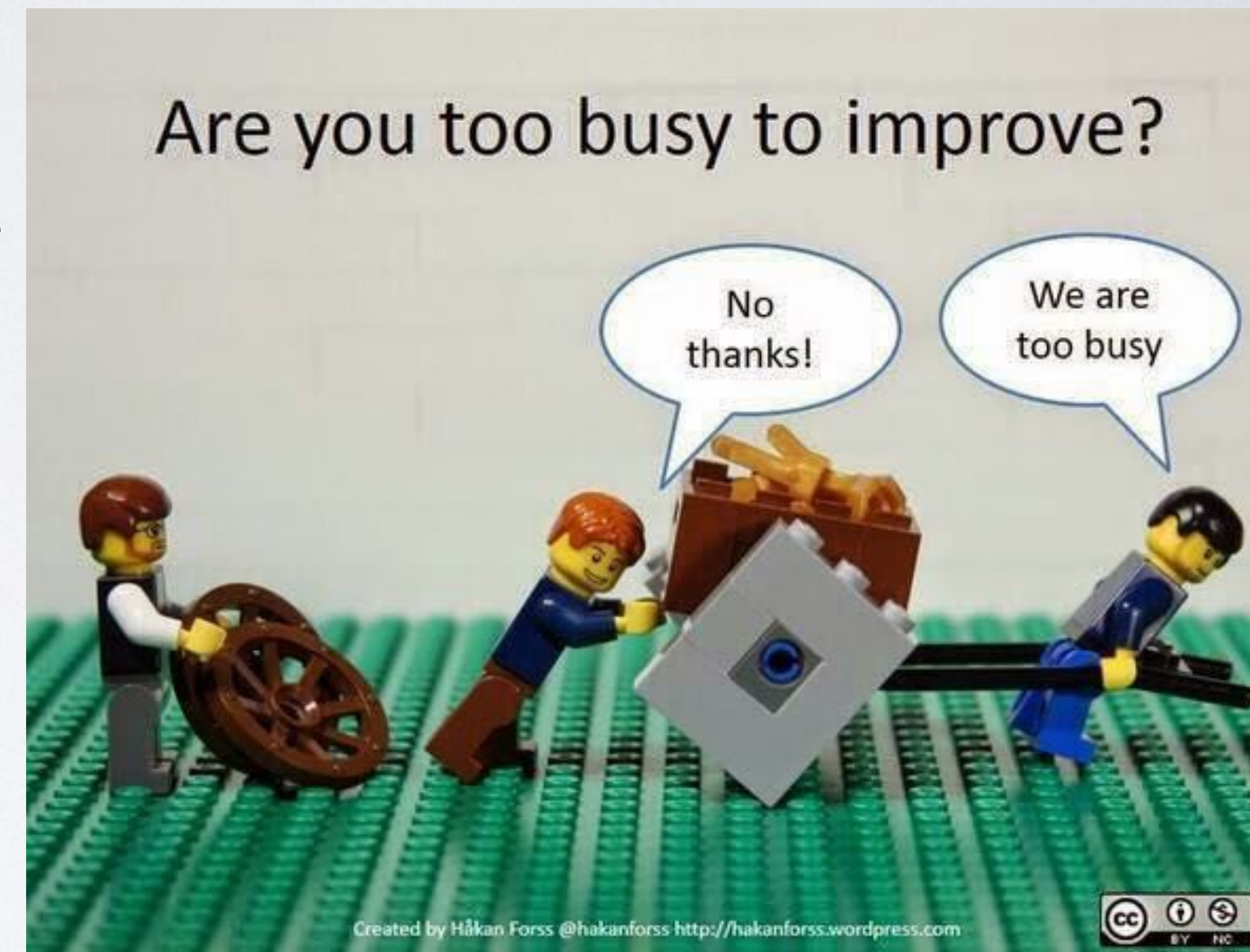






# EXTREME PROGRAMMING

- **Pratiques: Refactoring**
  - Restructuration et simplification permanente
  - Amélioration continue de la lisibilité
  - Amélioration continue de la maintenance
  - Rendre le code de plus en plus générique







# EXTREME PROGRAMMING

- **Pratiques: Pair Programming**

- Ecriture du code par paires de développeurs devant un ordinateur
- Implementation immédiate
- Application dans sa globalité
- Echanges des rôles fréquents et réguliers

