Tutoriels

ENSIIE - Projet informatique

2 mars 2018

Avertissement

Dans la suite se trouvent essentiellement des conseils, des manières de faire ou de ne pas faire quand on développe.

- Ce n'est pas la vérité absolue, il existe d'autres méthodes et d'autres outils.
- Lire ces explications ne vous dédouane pas d'aller lire la documentation ou d'autres tutoriel en ligne. Il n'y a ici que le strict minimum.

  Tout le tutoriel est sur `www.ensiie.fr/~dimitri.watel`

Plan

Plan

Exemple : factorielle.h

```c
typedef int nombre;
void fact(nombre n);
```

Exemple : factorielle.c

```c
#include "factorielle.h"
void fact(nombre n){
        if(n > 1)
                return fact(n - 1) * n;
        return 1;
}
```

Fichier `.h`

Un fichier `.h` est une *Interface* (ou *Header*) dans laquelle on indique :

- des prototypes de fonctions
- des définitions de types

Une interface contient uniquement les informations utiles à un utilisateur : quelles fonctions puis-je utiliser dans cette bibliothèque ?

Fichier .c

Un fichier `.c` est une *Source* dans laquelle on indique :

- des implantations des fonctions du `.h` associé
- des implantations de fonctions, de structures, de constantes, ... *internes*

Une source contient des informations inutiles à l'utilisateur, mais utile à un développeur qui souhaiterait rejoindre/maintenir le projet.

Pourquoi faire ?

Un utilisateur de `factorielle.h` n'a pas besoin de savoir comment elle est implantée. Il n'a pas besoin de `factorielle.c`.

```
#include <stdlib.h>
#include <stdio.h>
#include "factorielle.h"
int main(void){
        printf("%d\n", fact(3));
}
```

Pourquoi faire ?

Ainsi, on peut pré-compiler un fichier qui utilise `factorielle.h` alors que `factorielle.c` n'est pas compilé, voire pas codé.
Le code suivant renvoie une erreur, quelle que soit l'implantation.

```
#include <stdlib.h>
#include <stdio.h>
#include "factorielle.h"
int main(void){
        printf("%d\n", fact("abc"));
}
```

Compilation séparée

Compilation séparée

Si on sépare le code en différents modules/bibliothèques :

- on peut programmer séparément, sans attendre les implantations des autres
- on peut pré-compiler indépendamment les modules
- on peut réutiliser plus facilement certains modules
- on peut rédiger les tests des modules plus facilement

Pour pré-compiler : `gcc -Wall -Wextra -ansi -c main.c`
On obtient le fichier `main.o` (non exécutable).

Le linker : comment obtenir le fichier exécutable ?

Fichier .o

Un fichier `.o` est un fichier *Objet*. C'est une précompilation d'un fichier `.c` qui n'est pas éxécutable.
Il contient une version compilée de la source et quelles fonctions de quelles interfaces sont utilisées.

Il ne manque plus qu'à lui dire où sont les implantations des interfaces. C'est le rôle de *l'éditeur de lien* ou *linker*.

```
gcc -Wall -Wextra -ansi fact.o main.o -o testmain
```

Chaîne de compilation

- `gcc -Wall -Wextra -ansi -c main.c -o main.o`
- `gcc -Wall -Wextra -ansi -c fact.c -o fact.o`
- `gcc -Wall -Wextra -ansi fact.o main.o -o testmain`

Makefile

Makefile

Un Makefile est un fichier qui permet d'automatiser certaines opérations, en particulier :

- compilation d'un projet
- compilation d'une partie d'un projet
- nettoyage de fichiers temporaires
- archivage
- compilation puis exécution de tests

Pour simplifier grossièrement, il agit comme un script shell qui serait organisé pour n'exécuter que les commandes nécessaires à une opération spécifique.

Exemple

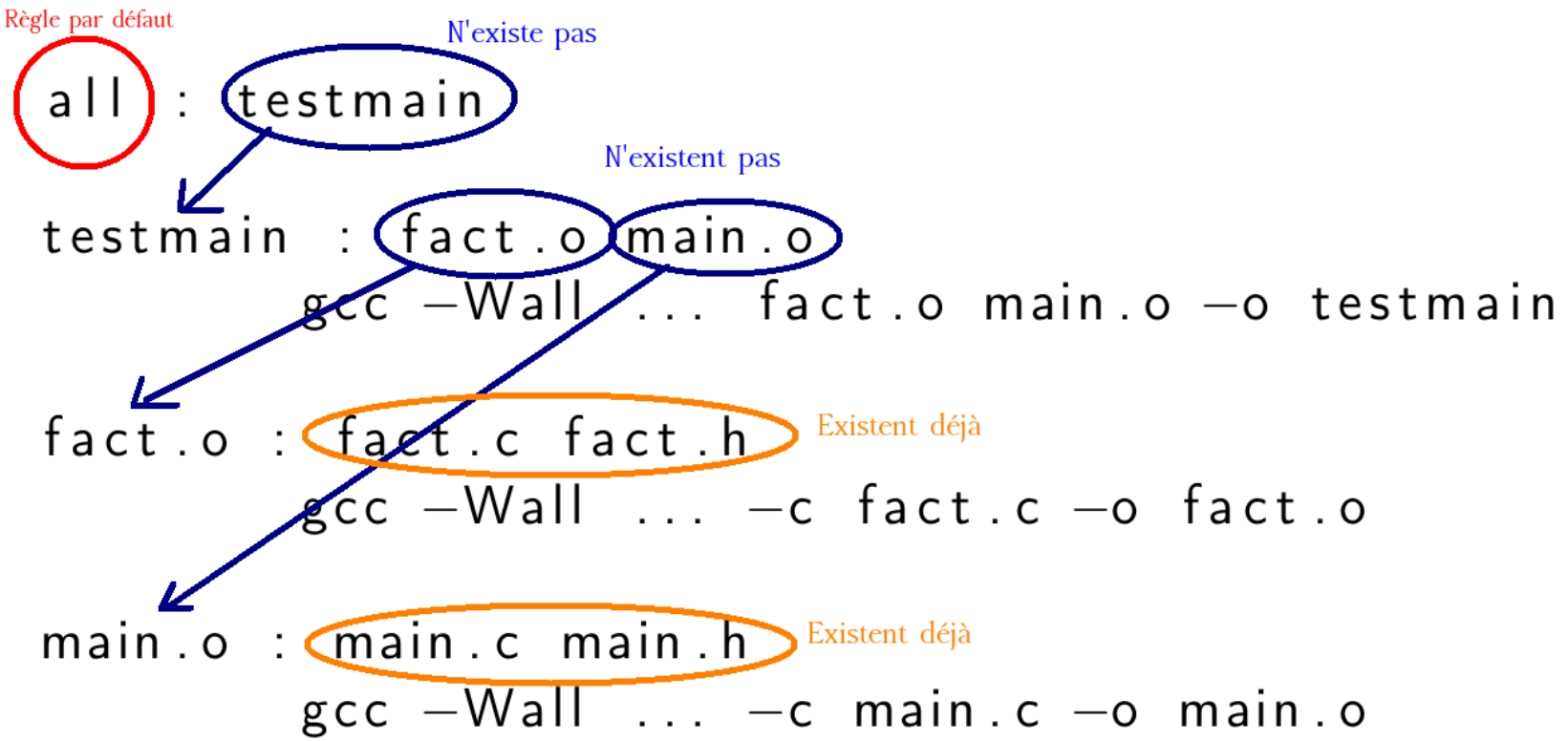
```
all : testmain

testmain : fact.o main.o
	gcc -Wall ... fact.o main.o -o testmain

fact.o : fact.c fact.h
	gcc -Wall ... -c fact.c -o fact.o

main.o : main.c main.h
	gcc -Wall ... -c main.c -o main.o
```

Exemple


Règle par défaut
N'existe pas
all : testmain
N'existent pas
testmain : fact.o main.o
gcc -Wall ... fact.o main.o -o testmain
fact.o : fact.c fact.h
Existent déjà
gcc -Wall ... -c fact.c -o fact.o
main.o : main.c main.h
Existent déjà
gcc -Wall ... -c main.c -o main.o

Exemple : variable

```
CC = gcc -Wall -Wextra -ansi

all : testmain

testmain : fact.o main.o
	$(CC) fact.o main.o -o testmain

fact.o : fact.c fact.h
	$(CC) -c fact.c -o fact.o

main.o : main.c main.h
	$(CC) -c main.c -o main.o
```

Exemple : mots-clefs

```
CC = gcc -Wall -Wextra -ansi

all : testmain

testmain : fact.o main.o
	$(CC) $^ -o $@

fact.o : fact.c fact.h
	$(CC) -c $< -o $@

main.o : main.c main.h
	$(CC) -c $< -o $@
```

Exemple : %

```
CC = gcc -Wall -Wextra -ansi

all : testmain

testmain : fact.o main.o
	$(CC) $^ -o $@

%.o : %.c %.h
	$(CC) -c $< -o $@
```

A retenir

Mots-clefs

Pour une règle dy type `cible : source1 source2 ... sourcen`

- `$@` = cible
- `$<` = source1
- `$^` = source1 source2 ... sourcen

Make

- `make` exécute le fichier nommé `./Makefile`
- `make -p` indique d'abord toutes les règles par défaut
- `make -n` indique uniquement les commandes qui seront exécutées, sans les exécuter
- `make -f file` utilise `./file` comme Makefile

Plan

Radide définition

Doxygen est un outil qui permet de gérer la documentation de son code. Il permet à terme de fournir une page web ou un pdf contenant la documentation du code que le développeur souhaite publier.

Que faut-il commenter ?

Il faut commenter tout ce qu'il vous semble utile de préciser pour qu'une personne extérieure puisse utiliser ou s'approprier le code.

- Commenter les fichiers headers pour les utilisateurs exétieurs
- Commenter les fichiers sources pour les autres développeurs

Commenter une fonction dans un .h

```
/**
* Calcule la factorielle d'un entier
*/
void fact(int n);
```

Commenter une fonction dans un .h

```
/**
* \brief Calcule la factorielle de \a n
* \param n Un entier positif
* \return La factorielle de \a n
*/
void fact(int n);
```

Commenter une fonction dans un .h

```
/**
* \brief Calcule la factorielle de \a n
* Calcule la factiorielle de l'entier positif
* non nul  \a n, c'est a dire le produit
* des entiers de 1 a n.
* \attention \a n doit etre un entier positif
* non nul.
* \param n Un entier positif
* \return La factorielle de \a n
*/
void fact(int n);
```

Quelques mots-clefs

Mots-clefs utiles.

- `\brief` : Décrire succintement la fonction. Une phrase maximum.
- `\detail` (mots-clef facultatif) : Décrire plus en détail la fonction.
- `\param k` : Décrire le paramètre d'entrée k de la fonction.
- `\return` : Décrire la sortie de la fonction.
- `\a` : Met le mot qui suit en italique
- `\b` : Met le mot qui suit en gras
- `\attention` : Prévenir le lecteur d'un point (très) important (comme les cas non couverts par la fonction).

Commenter la première ligne d'un fichier

```
/**
* \file geometry.h
*
* Ce fichier decrit un ensemble de fonctions
* geometriques utiles. Il contient 3 fonctions
* et un type:
* - le type \a point definit un point du plan
* - dist(p, q) pour calculer la distance entre deux
* points p et q
* - milieu(p, q) pour trouver le point milieu entre
* p et q
* - sym(p) pour calculer le symetrique de p par
* rapport a l'origine
*/
```

Créer la documentation (sous UNIX)

Créer le fichier de configuration

Pour créer la documentation, il faut configurer :

- `doxygen -g` crée un fichier nommé `Doxyfile`
- Ouvrez le fichier et changez les paramètres suivants :
  - `PROJECT_NAME`
  - `HAVE_DOT` : `NO`
  - `INPUT` : Dossiers où sont les sources commentées

Créer la documentation

`doxygen Doxyfile` crée 2 dossiers `html` et `latex` contenant respectivement la documentation au format html et latex.

Plan

Commande utile

Pour utiliser valgrind :

```
valgrind -leak-check=full -track-origins=yes file
```

Que regarder ? Si aucune erreur

```
All heap blocks were freed - no leaks are possible
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
```

Que regarder ? Si fuite mémoire

```
LEAK SUMMARY definitely lost: 4,000,004 bytes in 1
blocks
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0
from 0)
```

Comment la trouver ?

En ajoutant l'option -g à gcc lors de la compilation (avant d'utiliser valgrind), valgrind a plus d'infos :

```
4,000,004 bytes in 1 blocks are definitely lost in
loss record 1 of 1
at 0x4C2BBAF: malloc (vg_replace_malloc.c:299)
by 0x10890D: create_graph (graph_matrix.c:16)
by 0x10885D: main (main.c:36)
```

Autres erreurs possibles

On peut facilement trouver d'autres erreurs comme :

- lecture/écriture invalide (typiquement pour un tableau)
- utilisation de mémoire (tableau, pointeur) non initialisée
- libérer plusieurs fois un pointeur

Plan

Un bon test

Le bon test

Un *bon test* ne dépend pas des fichiers source, uniquement des interfaces. Un *bon test* peut donc être rédigé en même temps que les spécifications du projet. Un *bon test* doit être **en accord avec la documentation**.

Test unitaire

Un test unitaire ne teste qu'une partie atomique des spécifications sous des conditions précises qui, bien généralement, ne couvrent pas tous les cas.

Exemples

- Est-ce que `fact(3)` renvoie 6 ?
- Est-ce que `fact(0)` renvoie 1 ?

Cas non gérés

Attention, si la documentation considère un cas comme non spécifié, non géré, alors il ne doit pas être testé !
Par exemple,

- Est-ce que `fact(-1)` renvoie une exception ?

doit être testé s'il est explicitement écrit dans la documentation "Si n est négatif, fact(n) renvoie une exception".

Comment tester ?

CUnit

CUnit est une bibliothèqe de tests unitaires pour C. Il permet de programmer des tests, de les exécuter, et d'afficher un résumé des tests réussis ou échoués.

Comment tester ?

```
#include "CUnit/CUnit.h"
#include "CUnit/Basic.h"
#include "factorielle.h"
```

Comment tester ?

```
void test_la_factorielle_de_3_est_6(void)
{
        CU_ASSERT(fact(3) == 6);
}
```

ou

```
void test_la_factorielle_de_0_est_1(void)
{
        CU_ASSERT_EQUAL(fact(0), 1);
}
```

Comment tester ?

```
int main(void){
        [...]
        if(
        (NULL == CU_add_test(pSuite,
            "Classic_test,fact_of_3",
                test_la_factorielle_de_3_est_6))
         ||
         (NULL == CU_add_test(pSuite,
            "Test_of_fact_0",
                test_la_factorielle_de_0_est_1))
         )
         {[...]}
        [...]
}
```

Comment tester ?

Pour compiler

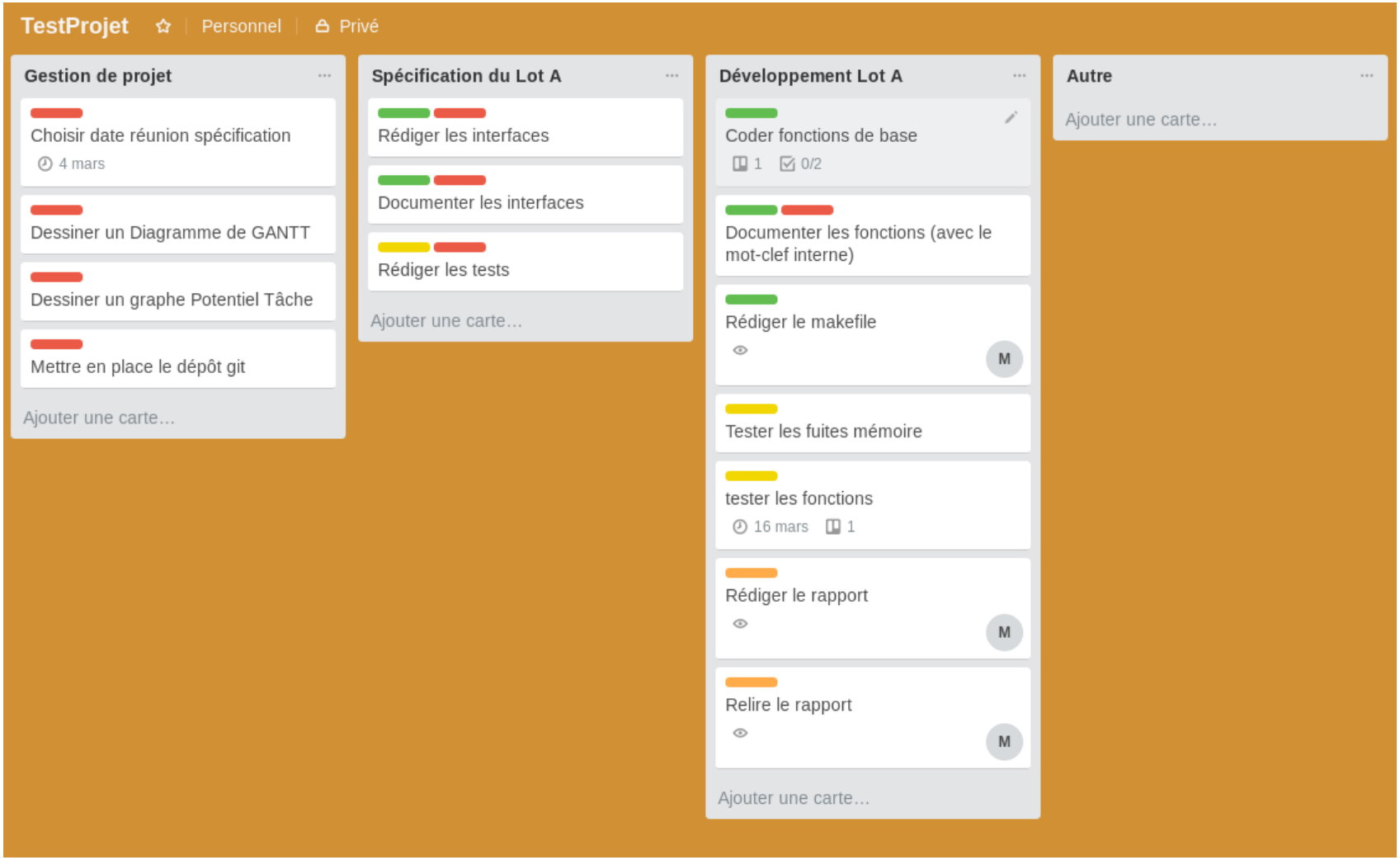
```
gcc -Wall -Wextra -ansi -lcunit
   test.c factorielle.o -o test
```

Plan

Présentation

Trello

Trello est un outil gratuit qui permet (entre autres) d'organiser ses tâches.


TestProjet
Personnel
Privé
Gestion de projet
Choisir date réunion spécification
4 mars
Dessiner un Diagramme de GANTT
Dessiner un graphe Potentiel Tâche
Mettre en place le dépôt git
Ajouter une carte…
Spécification du Lot A
Rédiger les interfaces
Documenter les interfaces
Rédiger les tests
Ajouter une carte…
Développement Lot A
Coder fonctions de base
1
0/2
Documenter les fonctions (avec le mot-clef interne)
Rédiger le makefile
M
Tester les fuites mémoire
tester les fonctions
16 mars
1
Rédiger le rapport
M
Relire le rapport
M
Ajouter une carte…
Autre
Ajouter une carte…

Plan

Gestionnaire de versions

Gestionnaire de versions centralisé

Un gestionnaire de versions centralisé consiste en un serveur qui contient *le dépôt*, c'est-à-dire toutes les versions du code depuis le début du projet.
Un développeur peut récupérer n'importe quelle version du code à partir du dépot et déposer une nouvelle version.

Un gestionnaire décentralisé est un système d'échange où chacun dispose des versions du projet qu'il souhaite avoir.

Git

Git

Git est un gestionnaire de versions décentralisé, (qu'on utilisera comme un gestionnaire centralisé).

Pour quoi faire ???? : chacun travaille à son rythme.

Cas d'utilisation

Tintin, Milou et Haddock travaillent sur le projet "BD".

- 10h02 : Tintin modifie le fichier `tome1.c`
- 11h08 : Milou modifie le fichier `tome2.c`
- 12h : Tintin envoie ses modifications.
- 12h14 : Haddock se connecte et récupère une nouvelle version
- 12h43 : Milou envoie ses modifiations
- 12h44 : Le serveur répond qu'il ne peut pas car il existe une nouvelle version qu'il n'a pas récupéré.
- 12h45 : Milou récupère la dernière version
- 12h46 : Milou envoie ses modifications.

Pour quoi faire ???? : gestion des conflits.

Cas d'utilisation

Tintin, Milou et Haddock travaillent sur le projet "BD".

- 10h02 : Tintin modifie le fichier `tome1.c`
- 11h08 : Milou modifie le fichier `tome1.c`
- 12h : Tintin envoie ses modifications.
- 12h45 : Milou récupère la dernière version et voit que Tintin a modifié le même fichier que lui.
- 12h46 : Milou regarde ce que Tintin a fait et fusionne ses modifications à celles de Tintin.
- 13h07 : Milou envoie ses modifications.

Lire sur un dépôt.

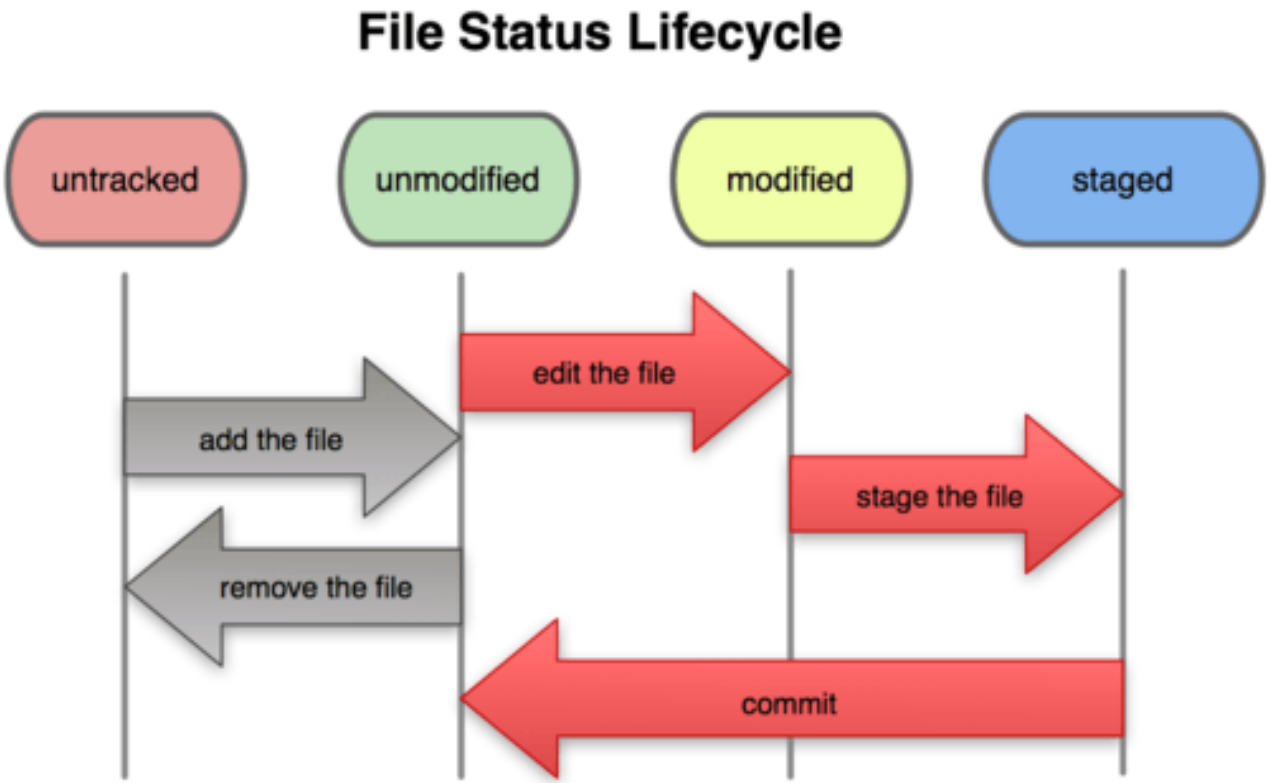
Commandes de lecture (simplifiée)

2 commandes :

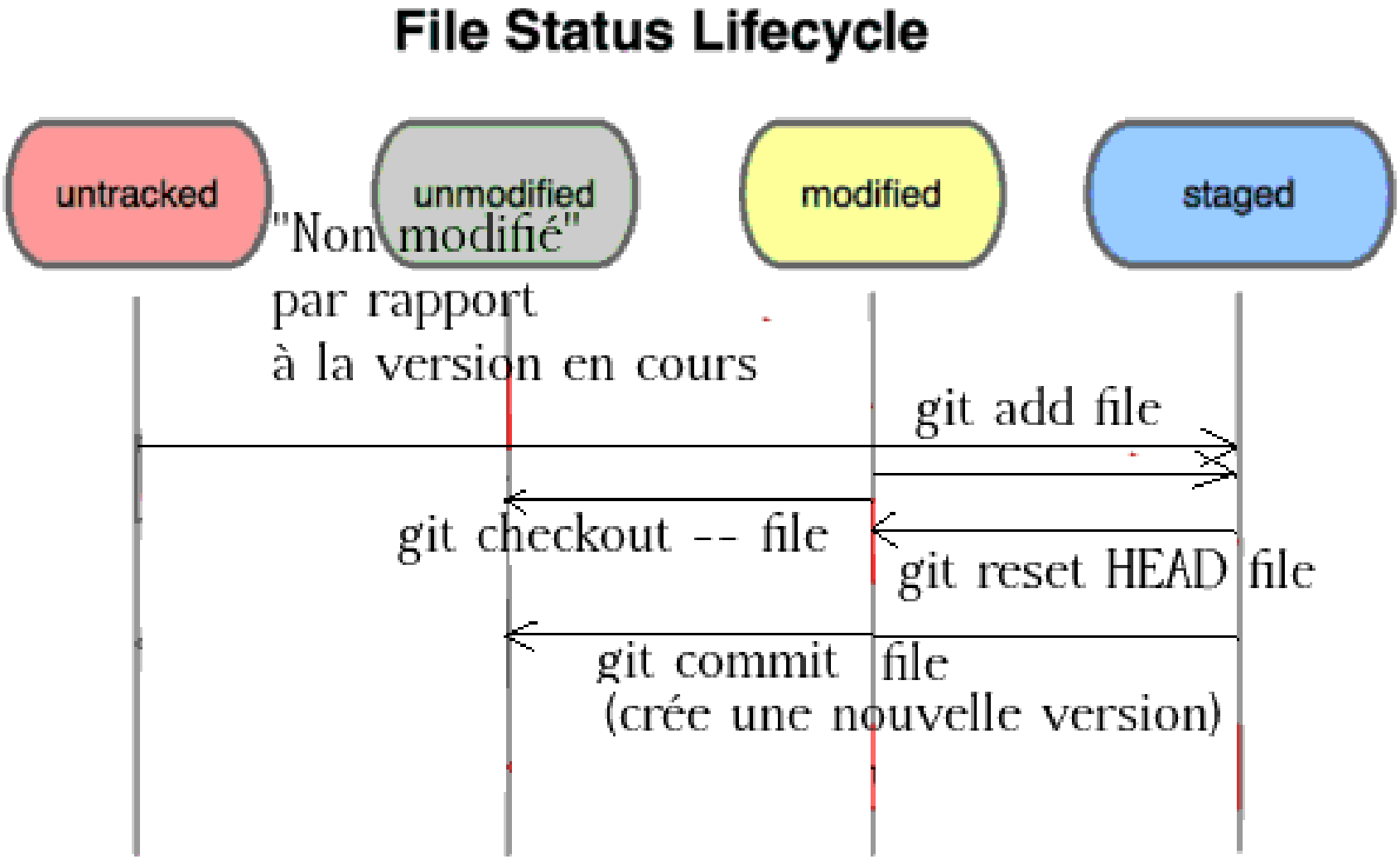
- `git clone https://addresse_du_depot` : va sur le serveur à l'adresse donnée, récupère le dépôt et copies toutes les versions sur sa machine.
- `git pull` : va sur le serveur cloné, récupère toutes les nouvelles versions du code qu'on a pas déjà copiées.

Etat local de son projet

`git status` : indique les fichiers qu'on a modifié depuis la dernière mise à jour `git diff file` : indique ce qui a été modifié dans `file` depuis la dernière mise à jour.



Écrire sur un dépôt.



Écrire sur un dépôt.

Commandes d'écriture (simplifiée)

2 commandes :

- `git commit` : crée une nouvelle version du code, avec toutes les modifications en état *Staged*
- `git push` : Envoie sur le dépôt toutes les versions du code commitée.

`git push` échoue si une autre personne a pushé avant vous. Il faut alors faire un `git pull`.

Gestion des conflits

`git pull` essaie de fusionner comme il peut toutes les modifications des autres développeurs. Des fois, c'est pas possible :
`Automatic merge failed; fix conflicts and then commit the result.`
`git status` indique alors quels fichiers doivent être fusionnés ensembles.

Gestion des conflits : ouvrir un fichier en conflit

Ouvrir un fichier non fusionné donnera des choses comme ça :

```
<<<<<<< HEAD:index.html
<div id="footer">contact : a@b.com
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> prob53:index.html
```

Il suffit de remplacer tout ces morceaux par le code souhaité : celui du dessus, celui du dessous ou un compromis. Ensuite, `git add`, `git commit` et `git push`.

Tagguer les commits

- `git tag -a machin -m "Commit machin"` ajoute le tag `machin` dernier commit effectué.
- `git tag` : liste les tags
- `git show machin:file` : affiche le contenu du fichier file de la version du commit `machin`
- `git diff machin file` : les modifications entre le fichier `file` du commit `machin` et le fichier `file` actuel.

Dernière commande sympa

```
git log -graph -pretty=format:"%Cred%h%Creset
%Cgreen(%cd) %C(bold blue)<%an>%Creset %s
-%C(yellow)%d%Creset
```

Documentation de git (en français)

`https://git-scm.com/book/fr/v2`

Ce document est très didactique. La partie 2, *Les bases de Git* vous permet d'être opérationnel assez vite. La partie 3 est utile mais sort du cadre du tutoriel.

Si vous êtes perdus

Ce qui suit est **mal** ! Il ne faut pas le faire.

Si vous avez fait n'importe quoi

Si vous avez cassé votre code en local ou si vous avez cassé le dépot, que vous n'arrivez plus à commiter. Une mauvaise solution consiste à copier quelque part tous les fichiers du projet que vous jugez être la dernière version à jour, puis à détruire le dépôt et à le recréer en y insérant la dernière version à jour.

Consignes

- Tout le monde doit coder un minimum
- On ne vous demande pas de maîtriser parfaitement tous les outils, juste de les tester, de les manipuler, d'en connaître les bases.
- Documentez toutes vos interfaces
- Documentez vos implantations avec le mots-clef `\internal`
- Montrez votre dépôt git à votre chargé à chaque séance
- Montrez votre compte Trello à votre chargé à chaque séance
- Rédigez un (petit) rapport entre chaque séances : le rapport narre ce qu'il s'est passé pendant la dernière séance et ce qui est prévu à la séance suivante. Ce n'est pas une documentation du code, c'est de la gestion de projet.