

INF3135

Construction et maintenance de logiciels

Chapitre 2: Outils de développement logiciel

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

1 Style de programmation

2 Documentation

3 Bats

4 Git

5 GitLab-CI

Style de programmation

Définition

Extrait de Wikipedia:

« Le style de programmation est un ensemble de règles ou de lignes directrices utilisées lors de l'écriture du code source d'un programme informatique. Il est souvent affirmé que suivre un style de programmation particulier aidera les programmeurs à lire et à comprendre le code source conforme au style, et aidera à éviter les erreurs. »

- **Conventions**, ensemble de **règles**
- Pour l'écriture de **code source**
- Améliore la **lisibilité**
- Permet de réduire les **erreurs**

Style de programmation en C

- C a été **standardisé** dans les années 80 (ANSI C89/C90)
- Mais aucun standard de **programmation** proposé

Quelques exemples

- Indian Hill
- NASA
- Noyau Linux (Linus Torvalds)
- GNU
- GNOME

« *The Single Most Important Rule* »

« *Check the surrounding code and try to imitate it.* »

— Extrait du site de GNOME

Contenu d'un fichier

- Un **module** C devrait contenir les éléments suivants, dans l'ordre:

```
/**  
 * Documentation d'en-tête du fichier  
 */  
  
#include // inclusion des bibliothèques  
  
#define // et autres constantes  
  
// Déclaration des types (struct, union, enum, typedef)  
  
// Déclaration des fonctions (avec leur docstring)  
// Regrouper les fonctions par thématique  
  
// Implémentation des fonctions (sans les documenter)  
  
// Fonction main
```

- **Bonne pratique:** utiliser des commentaires pour mettre en évidence la **structure** du fichier

Espacement

- Au plus **80 caractères** par ligne
- **Indentation**: 2, 4 ou 8
- **Tabulations** ou **espaces**: ne pas mélanger!
- **Aérer** autour des opérateurs et des délimiteurs:

```
for (int j = 3; j < 10; ++j) // Bien
for(int j=3;j<10;++j)        // À éviter
```

- Éviter les **suites** de lignes vides
- Aligner les **paramètres** lors d'un long appel de fonction

```
printf("L'équation\n  %.2lfx^2 %c %.2lfx %c %.2lf == 0\n",
      sol.equation.a,
      sol.equation.b >= 0 ? '+' : '-', fabs(sol.equation.b),
      sol.equation.c >= 0 ? '+' : '-', fabs(sol.equation.c));
```

Deux styles fréquents

Aéré:

```
if (valid)
{
    printf("Everything is fine\n.");
}
else
{
    printf("Something went wrong\n.");
}
```

Compact:

```
if (valid) {
    printf("Everything is fine\n.");
} else {
    printf("Something went wrong\n.");
}
```


Nomenclature (1/2)

Variables

- **Syntaxe** camelCase ou snake_case
- Plus la **portée** est importante, plus le nom devrait être **long**
- Préférer variables **courtes** pour indices d'un tableau: i, j, k

Types struct, union et enum

- **Syntaxe**: PascalCase ou snake_case
- Éviter typedef le plus possible

Orthographe

- Attention aux **fautes**
- Ne pas mélanger les **langues**

Nomenclature (2/2)

Fonctions

- **Syntaxe**: camelCase ou snake_case
- Si retourne void, utiliser verbe à l'**infinitif**: parse_values, initialize_canvas, multiply_arrays
- Si retourne nombre, utiliser **nom** correspondant: num_nodes, size, win_ratio, average_income
- Si retourne bool, utiliser verbe à l'**indicatif**: is_valid, has_attribute, contains_point
- Ne pas mettre **systématiquement** get et set

Fichiers

- **Syntaxe**: snake_case.c, snake_case.h
- Nom le plus **court** possible

Commentaires

- **Syntaxe:**

```
// Commentaire sur une ligne
```

```
/* Commentaire  
   multiligne */
```

```
/**  
 * Docstring  
 */
```

- *Docstrings* suffisent la plupart du temps
- **Commenter** le code traduit souvent un mauvais **découpage**
- Ou une mauvaise **nomenclature**
- **Éviter** de paraphraser le code
- **Supprimer** le code en commentaire à la livraison

Valeurs magiques

- Valeur magique = valeur constante
- **Nombres**, mais aussi **chaînes** de caractères
- À **éviter** le plus possible
- **Critère**: dès qu'elles sont utilisées plus d'une fois
- **Exemples**: dimensions d'un tableau, bornes de valeurs permises, messages d'erreur

Valeurs littérales acceptables

- 0 ou 1, très souvent
- 2 dans une formule mathématique ou pour vérifier la parité
- Le caractère '\0'
- Une chaîne fréquent ("yes", "no")
- Une option (-o|--output, -c|--count)

Factorisation

- Éviter au maximum la **duplication** de code
- Selon les **possibilités** du langage

Quand factoriser?

- Au fur et à mesure
- Ne pas attendre à la fin

Mécanismes

- À l'aide de **fonctions**
- **Généraliser** fonction en ajoutant paramètre
- **Réduire** nombre de paramètres en déclarant **types**
- À l'aide de l'opérateur **ternaire**
- À l'aide des **pointeurs** (on va y revenir)
- L'affichage et la lecture **formaté** (`printf`, `scanf`, `sscanf`)

Documentation

Plusieurs types de documentation

Code source (*docstrings*)

- **Modules**: description, auteurs, license, version, etc.
- **Fonctions**: description, paramètres, valeur de retour, etc.

Utilisateur

- Guide de l'utilisateur
- Souvent dans un fichier README
- Tutoriels pour l'utilisateur

Développeur

- Documentation des **modifications** apportées
- Guide du développeur
- Tutoriels pour le développeur

Langage de balisage léger

Définition (extraite de Wikipedia):

« Un langage de balisage léger est un type de langage de balisage utilisant une syntaxe simple, conçu pour être aisé à saisir avec un éditeur de texte simple, et facile à lire dans sa forme non formatée. »

Exemples:

- Markdown,
- ReStructuredText,
- AsciiDoc, etc.

Contre-exemples:

- HTML, XML: pas légers!
- YAML, JSON, légers, mais plutôt pour structurer des données

Markdown

- **2004**: créé par John Gruber avec Aaron Swartz
- Peu modifié ensuite par les auteurs originaux
- Extension de fichier: `.md` ou `.markdown`
- Peut facilement être transformé en HTML ou en PDF
- grâce notamment au programme **Pandoc**
- Supporté sur plusieurs **plateformes** ou **forums**
- Pas de **standardisation** formelle
- Possibilité d'**enchasser** du HTML
- **Attention!** éviter s'il existe un équivalent Markdown!

Plusieurs variantes (*flavors*)

- **Multimarkdown**, qui est une extension
- **GitHub Flavored Markdown**, d'abord développé pour GitHub
- **GitLab Flavored Markdown**, développé pour GitLab

Formatage

- **Emphase** (balise `` en HTML): étoiles simples `*mot*` ou soulignés simples `_mot_`
- **Emphase forte** (balise ``): étoiles doubles `**mot**`
- **Souligner**: soulignés doubles `__mot__`
- **Bout de code** (balise `<code>`): apostrophes inversées ``mot``
- **Citation** (balise `<blockquote>`): commencer par `>`

```
> Extrait d'une conversation qu'on souhaite commenter  
> Peut être sur plusieurs lignes
```

- **Paragraphes** (balise `<p>`): il suffit de laisser une ligne vide

```
Premier paragraphe
```

```
Deuxième paragraphe
```

Bloc de code

- Trois **apostrophes inversées** (*backticks*), suivi du **langage**

```
```c
#include <stdio.h>

int main(int argc, char *argv[]) {
 printf("Hello, world!\n");
 return 0;
}
```
```

```
```sh
$ sudo apt install pandoc
$ gcc -o maj maj.c
```
```

- **Sans langage** associé: indenter de 4 espaces

```
Bout de texte qui apparaîtra comme du code
Pour des extraits de fichiers texte, par exemple
```

Listes

- Liste **non ordonnée** (balise): étoiles ou tirets

- * Premier élément
- * Deuxième
 - * Élément imbriqué (au moins 4 espaces)
- * Troisième

- Liste **ordonnée** (balise): chiffre suivi d'un point

1. Premier élément
2. Deuxième élément
3. Troisième élément

- Liste **à cocher**: crochets avec x optionnel

- [] Finir TP1
- [] Relire notes de cours
- [x] Se reposer

Titres

- Pour **structurer** un document Markdown (balises <h1> à <h6>)

```
# Travail pratique 1
```

```
## Description
```

```
## Auteurs
```

```
## Exemples
```

```
### Exemple 1
```

```
### Exemple 2
```

- Attention de ne pas mettre **trop** de titres
- Ajuster la **profondeur** selon la taille du document

Liens

- Pour insérer un **hyperlien**:

```
[texte](lien relatif ou absolu)
```

- Pour insérer une **image**:

```
![texte](lien relatif ou absolu vers l'image)
```

Documentation

- Vers **sites officiels**
- Pour les **références** (Wikipedia, code, article, livre, etc.)

Mathématiques

- Supporté dans certaines **variantes**
- **Exemples:** Mattermost
- Ou encore GitLab Flavored Markdown
- **Dans le texte:** un dollar suivi d'une apostrophe inversée
- **Bloc mathématique:** comme pour le code, avec le mot `math`

Dans le texte, c'est comme ça: $(x + 1, y - 2)$.

Pour un bloc mathématique

```
```math
f(x,y) = x^2 + y^2 - 1
```
```

Documentation du code source

- Aucun standard de **documentation** officiel
- Selon le projet, le standard varie
- Dans le cours, nous allons utiliser le standard **Javadoc**

| Étiquette | Description |
|-------------|---|
| @author | Auteur du module ou de la fonction |
| @deprecated | Indique que la fonction ou le module ne devrait plus être utilisé |
| @exception | Décrit le type d'exception qui peut être soulevée |
| {@link} | Insère un lien vers un autre module, fonction, etc. |
| @param | Une brève description d'un paramètre de fonction |
| @return | Une brève description de la valeur de retour d'une fonction |
| @see | Indique une fonction ou un module relié |
| @version | Indique le numéro de version de la fonction ou du module |
| etc. | |

Documentation d'en-tête: *docstrings*

Fichier

- Description **générale** en une phrase (obligatoire)
- Description **détaillée** (optionnelle)
- **Exemples** d'utilisation (optionnels)
- **Auteur** (obligatoire)

Fonctions

- Description **générale** en une phrase (obligatoire)
- Description **détaillée** (optionnelle)
- **Exemples** d'utilisation (optionnels)
- Description de **chaque paramètre** (obligatoire)
- Description de la **valeur de retour**, s'il y en a une (obligatoire)

Examples

```
/**
 * Loads the company data from a JSON file
 *
 * @param company    The resulting company object
 * @param filename   The JSON filename path
 * @return           Error code indicating success or error
 */
enum error load_company_data(struct company *company,
                             const char *filename);

/**
 * Indicates if a triangle contains a given point
 *
 * @param t    The triangle
 * @param p    The point
 * @return     True if and only if the triangle contains the point
 */
bool triangle_contains_point(const struct triangle *t,
                             const struct point *p);
```

Documentation des modules

- Toujours documenter l'**en-tête des fichiers**:
- Utiliser le format **Markdown** (souvent reconnu)

```
/**
 * geometry.c
 *
 * Provides different data structures and functions for handling
 * 2-dimensional geometry.
 *
 * The basic type is `struct point`:
 *
 * ```c
 * struct point p = {1.5, -2.3};
 * print_point(&p);
 * ```
 *
 * [...]
 *
 * @author Alexandre Blondin Masse
 */
```

Bats

Bats

- *Bats* = *Bash Automated Testing System*
- **2011-2016**: créé et maintenu par Sam Stephenson
- **Ancien lien**: <https://github.com/sstephenson/bats>
- Depuis **2017**: maintenu par la communauté bats-core
- **Divergence** (*fork*) du projet original
- **Lien actuel**: <https://github.com/bats-core/bats-core>
- **Paquets**: Homebrew et NPM
- **Image**: DockerHub

Installation

```
# Avec apt - version pas à jour
$ sudo apt install bats

# À partir des fichiers sources - version récente
$ git clone https://github.com/bats-core/bats-core.git
$ cd bats-core
$ sudo ./install.sh /usr/local
```

Tests unitaires

Syntaxe

```
@test "Nom du test" {  
    # Suite de commandes shell  
    # Suite de tests (entre crochets [])  
}
```

Commandes et variables spéciales

- run: exécute et remplit les variables status, output et lines
- \$status: **code de retour** de la commande appelée
- \$output: **sortie** résultante (stdout et stderr combinés)
- \${lines[i]}: i-ème **ligne** de \$output
- skip: permet de **sauter** un test

Exemples

```
@test "Addition" {
    resultat="$(echo $((1 + 2)))" # Pas obligé d'utiliser run
    [ "$resultat" -eq 3 ]         # Teste si sortie de echo est 3
}

@test "Avec run" {
    run echo $((1 + 2))           # Avec run
    [ "$status" -eq 0 ]          # Teste code de retour de echo
    [ "$output" == "3" ]         # Teste si sortie de echo est 3
}

@test "Plusieurs lignes" {
    run echo -e "ligne 1\nligne 2" # Avec run
    [ "${lines[0]}" == "ligne 1" ] # Teste contenu de la 1re ligne
    [ "${lines[1]}" == "ligne2" ]  # Teste contenu de la 2e ligne
}

@test "Un test désactivé" {
    skip                          # On désactive le test
    run echo "un autre test"
    [ "$status" -eq 1 ]          # Teste si echo échoue
}
```

Invocation

Il suffit d'entrer la commande bats <fichier>:

```
$ bats tests.bats
✓ Addition
✓ Avec run
✗ Plusieurs lignes
  (in test file tests.bats, line 15)
    `[ "${lines[1]}" == "ligne2" ]    # Teste contenu de la 2e
      ligne' failed
- Un test désactivé (skipped)

4 tests, 1 failure, 1 skipped
```

Plusieurs options (bats --help):

- -c|--count: compter le nombre de tests
- -f|--filter: lancer tests qui vérifient une ER
- -F|--formatter: préciser le format d'affichage
- -t|--tap: afficher selon le protocole TAP
- -j|--jobs: tester en parallèle

Protocole TAP

- **TAP** = *Tests Anything Protocol*
- Format **texte** simple
- **Site officiel**: <http://testanything.org/>
- **Spécification**
- Format utilisé par **GitLab-CI** (pas de caractères spéciaux):

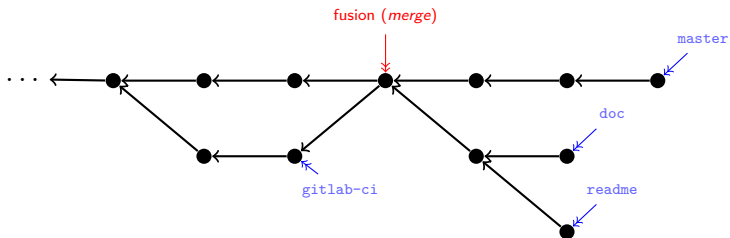
```
$ bats tests.bats --tap
1..4
ok 1 Addition
ok 2 Avec run
not ok 3 Plusieurs lignes
# (in test file tests.bats, line 15)
#   `[ "${lines[1]}" == "ligne2" ]    # Teste contenu de la 2e ligne
#   ' failed
ok 4 Un test désactivé # skip
```

- **Plan**: 1..4
- **Résultat**: ok ou not ok
- **Test ignoré**: skip
- **Commentaires**: avec #, etc.

Git

Historique d'un projet

Graphe orienté acyclique (DAG = *directed acyclic graph*)



4 types d'objets

- **Blob** = *Binary Large Object*: données binaires
- **Arbre**: arborescence de blobs (\approx arborescence de fichiers)
- **Commit**: référence à un arbre
- **Annotation**: référence vers un *commit*

Commit

Métadonnées

- **Message**: décrit les modifications apportées
- **Auteur** (*author*): auteur original
- **Date de création**: date, heure, fuseau horaire
- **Livreur** (*committer*): personne qui a intégré le *commit*
- **Date de livraison**: date, heure, fuseau horaire

Identification

- **Clé**: 40 caractères hexadécimaux
- Produite par l'algorithme **SHA-1** (*Secure Hash Algorithm*)
- Probabilité que la clé soit **unique** $\rightarrow 1$

Exemple

- Souvent, **auteur** = **livreur**, mais pas toujours
- `git show`: voir informations sur un *commit*
- `--quiet`: cacher *diff*, `--pretty=fuller`: toutes les métadonnées

```
$ cd bats-core
$ git show --quiet --pretty=fuller 3b33a5a
commit 3b33a5ac6afd7f01ff4120659e2a72b851081178
Merge: 7b032e4 eb120d9
Author:      Sam Stephenson <sstephenson@gmail.com>
AuthorDate:  Thu Oct 16 09:44:15 2014 -0500
Commit:      Sam Stephenson <sstephenson@gmail.com>
CommitDate:  Thu Oct 16 09:44:15 2014 -0500
```

Merge pull request #76 from jwerle/patch-1

Update package.json

```
$ git show --quiet --pretty=fuller 3be8246 | head -n 5
commit 3be82466a7355b3a6f40f428d8c6520b63241593
Author:      Henrique Moody <henriquemoody@gmail.com>
AuthorDate:  Wed Oct 30 22:10:00 2013 -0200
Commit:      Ross Duggan <rduggan@engineyard.com>
CommitDate:  Wed Aug 13 14:32:35 2014 +0100
```

Références

Référence courante

- HEAD: référence vers l'état courant
- HEAD~n: référence vers le n-ième *commit* parent
- detached HEAD: référence à un *commit* non pointé par une branche

Branches

- **Référence** nommée vers un *commit*
- master: branche « principale » qui pointe vers le **1er** *commit*

Références distantes (*remote*)

- **Dépôt distant** nommé origin par défaut
- origin/master: branche « principale » du dépôt distant

Création d'un dépôt

Nouveau dépôt

```
git init
```

Copie d'un dépôt existant

- **Commande:** `git clone`
- *Fork*: gérée par l'hébergeur (GitLab, Github), pas par Git
- Deux **protocoles**: HTTPS et Git (SSH)

Répertoire spécial `.git`

- **Décentralisé**: tout l'historique y est contenu
- Chaque dépôt Git est **équivalent**
- Le répertoire qui contient le sous-répertoire `.git` peut donc être **déplacé** n'importe où

Consulter l'historique

Commandes fréquentes

- `git log`: journal détaillé des modifications
- `git log --graph --all --color`: historique en graphe
- `git show`: voir un *commit* spécifique
- `git diff`: différence entre deux versions

Astuce dans `.gitconfig`

Ajouter un synonyme (*alias*) pour la commande `git gr`:

```
[alias]
  gr = log --graph --full-history --all --color --pretty=tformat
      : "%x1b[31m%h%x09%x1b[32m%d%x1b[0m%x20%s%x20%x1b[33m(%an)%
      x1b[0m"
```


Exemple

```
$ cd bats
$ git gr
* c750877      (origin/issue-290-debian-fix) build: test parallel invocation in [...]
* 7f0b346      (HEAD -> master, origin/master, origin/HEAD) Merge pull request [...]
|\
| * 3e9fd9d    test: filter parallell warnings in --job ordering test (Andrew Martin)
| * 17ff3f1    fix: parallel invocation for debian (Martin Schulze)
* | 743b02b    Merge pull request #310 from dimo414/patch-2 (Andrew Martin)
|\ \
| * | aeeb090   Remove TODO leftover from 118391d8e (Michael Diamond)
|/ /
* | c648a85    Merge pull request #291 from dimo414/patch-2 (Andrew Martin)
|\ \
| * | 220bb9d   docs: add to README `load` semantics (Michael Diamond)
* | | 9b0a9a5   Merge pull request #304 from dimo414/patch-3 (Andrew Martin)
|\ \ \
| | _|_/
|/| |
| * | 0b57bca   Remove "See the Background section above" link (Michael Diamond)
|/ /
* | b615ed8    Merge pull request #297 from martin-schulze-vireso/issue-290 [...]
|\ \
| * | 0deffcc   fix: parallel output (Martin Schulze)
|/ /
* | 90ce858    Merge pull request #296 from martin-schulze-vireso/issue-292 [...]
|\ \
| * | 338226a   fix: handle skip in teardown more gracefully (Martin Schulze)
| * | 0ec52d7   test: add failing test cases for `skip` (Andrew Martin)
| | /
* | 3a0d0d2    Merge pull request #288 from waterkip/fix-greadlinky (Andrew Martin)
[...]
```

État d'un projet

États possibles

- **Propre** (*clean*): aucun fichier versionné n'a été modifié
- **Modifié**: il y a eu certaines modifications
- **Indexé** (*staged*): certaines modifications ont été indexées

Connaître l'état d'un projet

```
$ git status      # Affichage long  
$ git status -s   # Affichage compact
```

Astuce dans `.gitconfig`

Ajouter un synonyme pour `git st`:

```
[alias]  
  st = status -s
```

Changer l'état du projet

Commande

```
git checkout
```

Astuce dans `.gitconfig`

```
[alias]
  co = checkout
```

Astuce dans `.bashrc`

```
# Retrieves current branch
function parse_git_branch_and_add_brackets {
  git branch --no-color 2> /dev/null | sed -e '/^[^*]/d' -e 's/*
    \(.*\)/\[\1\]/'
}
# Changes prompt
PS1="\e[36m\$(parse_git_branch_and_add_brackets)\e[32;1m\u\e[0m \e
  [33;1m[\w]\e[0m\n$ "
```

Exemples

```
$ git co master          # Se placer sur la branche principale
```

```
$ git co c648a85         # Se placer au commit c648a85
```

```
Note: checking out 'c648a85'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

```
$ git co origin/master  # Branche principale du dépôt distant
```

```
Note: checking out 'origin/master'.
```

You are in 'detached HEAD' state. You can look around, make experimental [...]

- Possible d'**ajouter** des *commits*
- Et de créer des nouvelles **branches**
- Ou de **déplacer** des branches (*rebase*)
- On va y revenir plus tard

Préparer une sauvegarde (*commit*)

Cycle de développement de base:

1. `git st`: vérifier que l'état est **propre**
2. Apporter des modifications au projet (tâche **atomique**)
3. `git st`: vérifier l'état des fichiers modifiés
4. `git diff`: vérifier les modifications
5. `git add`: indexer les modifications **ou**
`git add -p|--patch`: indexer des morceaux de modifications
6. `git commit`: valider la sauvegarde **ou**
`git ci` (en ajoutant l'*alias* `ci = commit`)

Raccourci

- `git ci -a`: combiner l'indexation et la validation
- De **toutes** les modifications

Message de *commit*

Les 7 règles d'un bon message (plus de détails)

1. Limiter le sujet à **50 caractères**
2. Séparer le **sujet** du **corps** par une ligne vide
3. Commencer le sujet par une **majuscule**
4. Ne pas terminer le sujet avec un **point**
5. Utiliser l'**impératif** dans le sujet
6. Limiter à **72 caractères** la largeur du corps
7. Dans le corps, décrire **quoi?** et **pourquoi?** plutôt que *comment?*
 - Ne pas **mélanger** les langues

Conventions

- Dans un projet existant, respecter **les conventions**
- **Exemple:** **commits conventionnels** (issu du projet Angular)

Réinitialiser l'état du projet

Fichier spécifique

`git checkout <fichier>`: annuler les modifications apportées à <fichier> depuis le dernier *commit*

Annuler l'indexation

`git reset`: annule les commandes `git add` précédentes

Annuler les modifications

- `git reset --hard`: restaure les fichiers avant modifications
- **Attention**: on ne peut pas revenir en arrière
- **Vérifier** avec `git diff` avant

Corriger un *commit*

Corriger le message

- `git ci --amend`
- Puis on réécrit le message

Corriger le contenu du dernier *commit*

- Apporter les **modifications** souhaitées normalement
- Puis faire `git ci --amend` plutôt que `git add`

Attention

- **Réécrit** l'historique
- À éviter si vous avez **partagé** des modifications

Récupérer et partager des modifications

Télécharger des historiques distants

```
git fetch
```

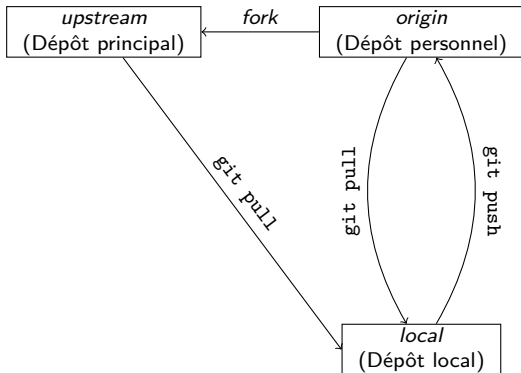
Fusionner deux historiques

- `git merge`: fusionne deux branches
- Peut entraîner des **conflits**
- On va y revenir
- `git pull`: tirer des modifications
- `git pull`: `git fetch` + `git merge`

Partager des modifications

- `git push`: « pousser » des modifications
- Doit souvent être précédé de `git pull` (conflits potentiels)

Se synchroniser avec des dépôts distants



- `git remote -v`: voir les informations sur dépôts distants
- `git remote add upstream <URL>`: ajouter un dépôt distant nommé *upstream*
- **Synchronisation** avec *upstream* sur master

GitLab-CI

Intégration continue

- En anglais, *continuous integration* (CI)
- Une **modification** à un logiciel ne devrait pas entraîner de régression
- Garantir la **compilation** (*build*)
- Mais aussi le **fonctionnement** (tests unitaires)
- Accepter seulement si **aucun test** n'échoue

Logiciels

- **Jenkins**: initialement pour Java, mais supporte plusieurs autres langages
- **Travis CI**: intégré directement à Github
- **GitLab CI**: intégré directement à GitLab

GitLab CI

- Documentation: <https://about.gitlab.com/gitlab-ci/>
- Tutoriel introductif: [ici](#)
- Mise en place: ajout d'un fichier nommé `.gitlab-ci.yml` qui respecte le format YAML et qui indique comment lancer les tests
- Lancés dans un « carré de sable » (*sand box*)
- Ce carré de sable est une **image Docker**
- Plusieurs images sont disponibles **par défaut**, mais vous pouvez aussi fournir vos propres images

Mise en place

- Ajout d'un fichier de configuration
- Et de scripts de vérification
- **Exemple:** lancement de la commande `make test`

Exemple

```
# Mise à jour de apt et installation de Bats
before_script:
  - apt-get update -qq
  - git clone "https://github.com/bats-core/bats-core.git" /tmp/bats
  - mkdir -p /tmp/local
  - bash /tmp/bats/install.sh /tmp/local
  - export PATH="$PATH:/tmp/local/bin"

# Pour vérifier la compilation
build:
  stage: build
  script:
    - make

# Tests unitaires
test:
  stage: test
  script:
    - make test
```

- Possible de spécifier l'**image** Docker
- Structuration des *pipelines*
- Peut conserver les **résultats** (artéfacts)
- On va y revenir