

# Techniques de Développement

## Outils et méthodes de développement

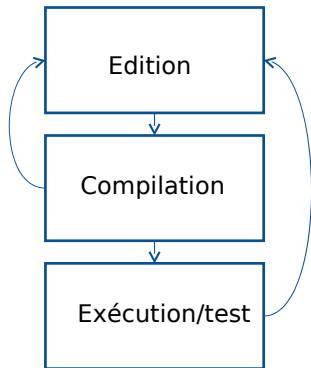


Licence de sciences mention informatique

2016

# Faciliter le processus de développement

Mise au point : erreurs syntaxiques et de comportement.



## Pour faciliter la correction et l'amélioration :

- Debugger.
- Vérificateurs d'allocation et d'accès mémoire.
- Analyseur de performances (profiler).

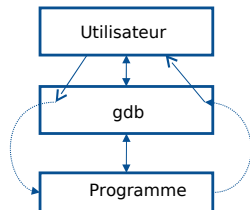
## Pour faciliter l'implémentation :

- Environnements de développement.
- Systèmes de gestion de versions.

- 1 GNU Debugger
- 2 Vérificateurs de la mémoire
- 3 Gestion de versions
- 4 Environnements de Développement Intégrés
- 5 Documentation
- 6 Licence Logiciel
- 7 Archivage

# GNU Debugger : GDB

- Compiler le programme avec l'option `-g` (ajout d'informations dans l'exécutable)
- Lancer le programme avec `gdb <prgm>`
- Donner des instructions à gdb.



Le debugger utilise les informations ajoutées à l'exécutable pour s'intercaler entre l'utilisateur et l'exécutable.

Il permet à l'utilisateur de guider l'exécution (lignes de codes exécutées une par une, sauts, etc.) et d'afficher des informations sur l'état du programme (variables, état de la pile, etc.) qu'il peut modifier à la demande.

# GDB : utilisation

## Modes d'utilisation :

- En ligne de commande (interpréteur).
- Avec une interface graphique (par ex. ddd ou dans un IDE).

## Opérations possibles :

- Stopper un programme lors de conditions spécifiées.
- Examiner ce qui s'est passé quand le programme s'est arrêté.
- Corriger un bug (à la volée ou recompiler le programme).

# GDB : commandes

Toutes les commandes suivantes peuvent être appelées à chaque interruption de l'exécution.

## Affichage du code :

`list [<nom fonction>|<num ligne>]` affiche 10 lignes du code source centrées sur la fonction, ou sur ligne num, ou autour de la ligne courante (si pas d'arguments).

## Exécution :

- `run [liste d'arguments]` (ou `r`) lance l'exécution le programme (avec la liste arguments).
- `kill` tue le programme en cours de mise au point.
- `quit` quitte le debugger.
- `help [sujet]` documentation en ligne.

# GDB : commandes

## Arrêt de l'exécution

- `break [fichier:] [<nom fonction>|<num ligne>]` (ou `b`) place un point d'arrêt. Gdb interrompt l'exécution lorsqu'il rencontre un point d'arrêt.
- `watch <variable>` (la variable doit être définie dans le contexte courant). Gdb interrompt l'exécution chaque fois que la variable est lue.
- `info breakpoints` liste les points d'arrêts.
- `disable` ou `enable breakpoints <num breakpoint>` désactive ou réactive le point d'arrêt.
- `clear [<nom fonction>|<num ligne>]` ou `delete <num breakpoint>` supprime le point d'arrêt.

# GDB : commandes

## Guidage interactif de l'exécution (après un arrêt)

- `continue` ou `c` : continuer l'exécution du programme jusqu'au prochain point d'arrêt.
- `step [<num>]` ou `s` : passer à la ligne suivante (ou aux num lignes suivantes), *en entrant* dans les sous-fonctions.
- `next [<num>]` ou `n` : passer à la ligne suivante (ou aux num lignes suivantes), *sans entrer* dans les sous-fonctions.
- `finish` : terminer l'appel de la fonction courante.



# GDB : commandes

## Inspection de l'environnement

- `backtrace` ou `bt` ou `where` : affiche la pile d'appels.
- `frame [<num>]` ou `f` : place le contexte à num (numéro de frame dans la pile d'appels) ; affiche le contexte courant si pas d'argument.
- `print [<variable>|<expression>|<variable>=<valeur>]` ou `p` : affiche la valeur de la variable / l'expression, calculée d'après le contexte courant. Permet aussi de modifier la valeur affectée à une variable.
- `x <variable>` : affiche l'adresse d'une variable.
- `display <expression>` : affiche le résultat de l'évaluation de l'expression à chaque arrêt.
- `undisplay <expression>` : annule l'affichage de l'expression.

# GDB : exemple (prog.c)

**démo**

```
1  #include "stdio.h"
2
3  void affiche(char *message)
4  {
5      int i = 3;
6
7      do {
8          printf("%c", (*message)+i);
9      } while ((*++message));
10     printf("\n");
11 }
12
13 int main()
14 {
15     char * mess1 = NULL;
16     char * mess2 = "Hello world.";
17     affiche(mess2);
18     affiche(mess1);
19     return 1;
20 }
```

## Compilation

```
$> gcc -g -o prog prog.c
```

## Exécution hors gdb

```
$> ./prog
```

```
Khoor#zruog1
```

```
Erreur de segmentation
```

# GDB : exemple (exécution simple dans gdb)

démon

```
$> gdb ./prog
```

```
(gdb) run
```

```
Starting program: ~/prog
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0x0804835b in affiche (message=0x0) at prog.c:8
```

```
8  printf("%c", (*message)+i);
```

```
(gdb) backtrace
```

```
#0 0x0804835b in affiche (message=0x0) at prog.c:8
```

```
#1 0x080483c3 in main () at prog.c:18
```

```
(gdb) c
```

```
Continuing.
```

```
Program terminated with signal SIGSEGV, Segmentation fault.
```

```
The program no longer exists.
```

# GDB : exemple (exécution guidée dans gdb)

démon

```
(gdb) break main
```

```
Breakpoint 1 at 0x804839c:
```

```
file prog.c, line 15
```

```
(gdb) run
```

```
Starting program: ~/prog
```

```
Breakpoint 1, main () at prog.c:15
```

```
15 char *mess1 = NULL;
```

```
(gdb) next
```

```
16 char *mess2 = "Hello, world.";
```

```
(gdb) print mess1
```

```
$1 = 0x0
```

```
(gdb) next
```

```
17 affiche(mess2);
```

```
(gdb) next
```

```
18 affiche(mess1);
```

```
(gdb) step
```

```
affiche(message=0x0) at prog.c:5
```

```
5 int i = 3;
```

```
(gdb) step
```

```
8 printf("%c", (*message)+i);
```

```
(gdb) step
```

```
Program received signal SIGSEGV,  
Segmentation fault.
```

```
0x0804835b in affiche (message=0x0)  
at prog.c:8
```

```
8 printf("%c", (*message)+i);
```

```
(gdb) print (*message)+i
```

```
Cannot access memory at address 0x0
```

- 1 GNU Debugger
- 2 Vérificateurs de la mémoire
- 3 Gestion de versions
- 4 Environnements de Développement Intégrés
- 5 Documentation
- 6 Licence Logiciel
- 7 Archivage

# Analyse de la mémoire

**Objectif :** détecter les erreurs d'allocation, de désallocation et d'accès mémoire. Ces erreurs sont souvent difficiles à détecter mais provoquent des résultats incorrects ou des interruptions d'exécution.

**Exemples courants :**

- Pointeurs fous (dangling pointer) dûs à un débordement d'un bloc mémoire alloué (par ex. accès à `tab[-1]`), l'oubli d'un caractère nul final d'une chaîne, ou l'utilisation d'une zone mémoire déjà libérée.
- Fuite de mémoire (memory leak) due à un espace mémoire inutilisé mais non libéré. L'exécutable grossit inutilement, voire il remplit toute la mémoire et plante ("Not enough memory").

Il existe différents outils d'analyse, qui travaillent de manière différente, par exemple :

## **Valgrind :**

- Vérifie les accès mémoires à l'exécution.
- Utilise les informations générées par l'option de compilation -g.
- Fait un bilan des mallocs / free.

## **Electric Fence :**

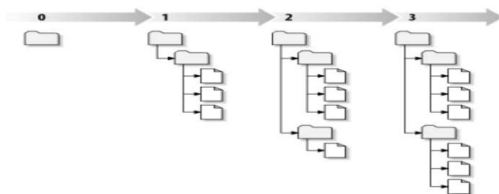
- Redéfinit malloc, free, etc.
- Sous forme d'une bibliothèque (compiler avec -lefence).
- Stoppe le programme à l'instruction erronée.

- 1 GNU Debugger
- 2 Vérificateurs de la mémoire
- 3 Gestion de versions**
- 4 Environnements de Développement Intégrés
- 5 Documentation
- 6 Licence Logiciel
- 7 Archivage



# Gestion de versions

## Pourquoi plusieurs versions d'un fichier ?



- Pour revenir en arrière (à une version stable) et faire des corrections.
- Pour conserver un historique de toutes les opérations (listes des modifications, qui a modifié, quand, différences entre versions, etc.).
- Indispensable pour le travail en équipe (verrous, gestion des conflits).
- Pour travailler en parallèle sur plusieurs branches.
- Pour garantir la sécurité (intégrité, disponibilité, confidentialité).
- Pour du code mais aussi un site web, de la doc, etc.

**Diff** est un utilitaire linux qui compare deux fichiers et produit un fichier "patch" contenant la différence entre les deux fichiers (lignes ajoutées ou supprimées).

`version2 - version1 = patch`

**Patch** est un utilitaire linux qui applique un patch à un fichier et génère le nouveau fichier.

`version1 = version2 + patch`

**Intérêt** : dans certain cas, il est plus avantageux de sauvegarder les modifications du fichier que N copies d'un fichier.

# Vocabulaire

**Dépôt (repository)** : c'est un répertoire partagé par tous, qui conserve l'historique des modifications.

**Révision** : chaque fichier à un numéro de version unique. Une incompatibilité d'interfaces doit entraîner un changement majeur. Les mineurs correspondent plutôt à des évolutions compatibles.

**Branches** qui permettent de :

- corriger un problème sur une ancienne version,
- développer 2 idées en parallèle (branche stable, branche testing, ...),
- gérer sa propre version d'un logiciel,
- fusionner après une divergence.

**Tags** : ce sont des marques symboliques sur une révision. Elles permettent de définir les versions et de nommer des branches (par ex. versions alpha, beta).

# Modèles de fonctionnement d'un sgv

## Différentes familles

**Local** Fonctionne dans un système de fichiers local, sans réseau.  
Exemples : SCCS, RCS.

**Client/Serveur (ou centralisé)** Un serveur centralise le dépôt, accessible à distance.  
Exemples : CVS, Subversion (SVN)

**Distribué (ou décentralisé)** Multiples copies du dépôt, branches locales.  
Exemples : bitkeeper, monotone, arch, darcs, mercurial, git, bazaar.

# Historique

## **SCCS (1972)** (local)

- SGV historique d'Unix.
- Définit les concepts de base de beaucoup de SGVs.
- Limites : verrouillage très strict, pas de fusion.

## **GNU RCS (1982)** (local)

- Extension de SCCS.
- Nouveautés : notion de fusion.
- Limites : gestion des branches très lourde, un seul utilisateur à la fois, une seule copie de travail.

# Historique

## CVS (1990) : **Concurrent Version System** (centralisé)

- Basé sur RCS.
- Nouveautés : Centralise le dépôt, autorise plusieurs copies de travail concurrentes, mode client/serveur simple.
- Limites : ne gère pas l'authentification (nécessite un compte Unix par utilisateur sur le serveur), travaille fichier par fichier, pas de commit atomique, ne gère pas les renommages ou les déplacements de fichiers, gestion des branches très lourde, mal adapté pour des développements parallèles.
- Amélioration : openCVS.

# Historique

## SVN (2000) : **subversion** (centralisé)

- Successeur de CVS.
- projet de Apache depuis 2010
- Nouveautés : commit atomique, renommage de fichiers, gestion des meta-données (droits d'accès, propriétaire), meilleure gestion des branches, stockage sophistiqué (base de données), accès distants via HTTP/DAV, interfaces graphiques.
- Limites : pas de mémoire des fusions.

# Historique

## BitKeeper...GIT (2005) : **subversion** (décentralisé)

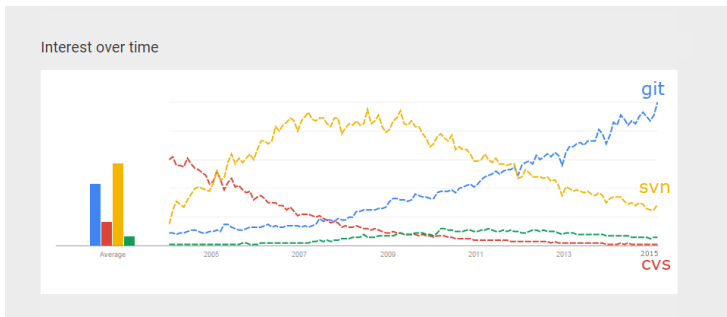
- Arrivée plus tardive des systèmes décentralisés.
- Permet d'être moins dépendant d'un serveur : travail en autonomie, déconnecté ou désynchronisé.
- Plateformes web : ex. GitHub.



# Historique

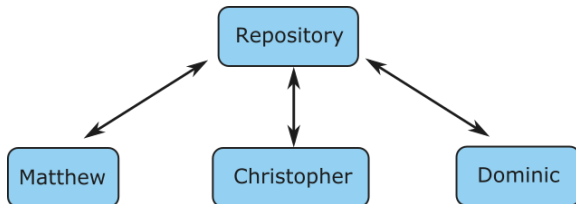
**SVN et GIT sont actuellement les plus utilisés** (décentralisé)

- GIT : environ 12 millions de développeurs (2016)
- GIT plus utilisé que SVN



Source Google Trends. Version interactive : <http://goo.gl/2xDMxg>

# Modèle client/serveur



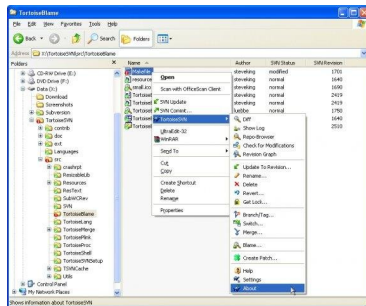
- Dépôt stocké dans un endroit partagé : par le système de fichiers ou par un mécanisme réseau (rsh/ssh ou protocole dédié).
- Plusieurs copies de travail en parallèle : opérations de fusion.
- Nécessaire d'avoir la connexion au dépôt pour "commiter".
- Le "tronc" a une importance particulière : modèle très centralisé.

# Modèle client/serveur : utilisation

## Travailler à plusieurs

- Pas de verrou sur les sources : chacun à sa propre copie.
- Gestion des conflits :
  - **Pas de commit sans update préalable.**
  - Si pas de conflit détecté : fusion automatique.
  - Si conflits détectés : résolution par les développeurs.
    - Pas de nouveau commit avant résolution du conflit.

- Visualisation de l'historique sous diverses formes.
- Annotation du code avec les contributions.
- Interface.



# SVN : commandes principales

## Administration (`svnadmin <commande>`)

- `create` : création du dépôt.

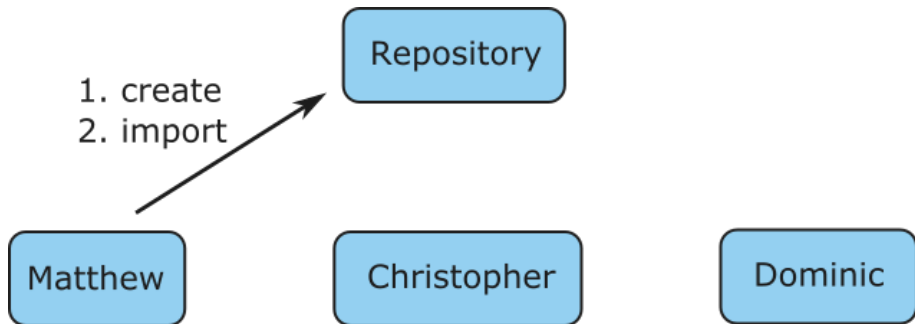
## Coté serveur

- `import` : introduire dans le dépôt.
- Définition des utilisateurs, de leurs droits, etc.

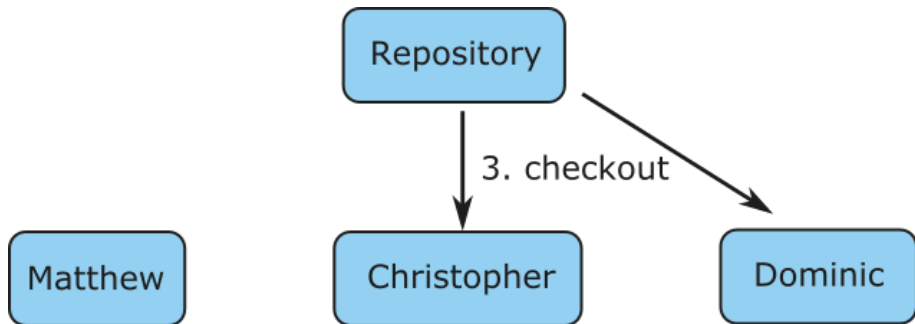
## Coté client (`svn <commande> [options]`)

- `checkout` (ou `co`) : récupérer une copie de travail du projet à partir du dépôt.
- `update` (ou `up`) : mettre à jour le projet.
- `commit` (ou `ci`) : archiver dans le dépôt.
- `status` : se comparer au dépôt.
- `add` : ajouter des documents.
- `remove` (ou `rm`) : supprimer des documents.
- Créer/fusionner des branches, étiqueter des versions.

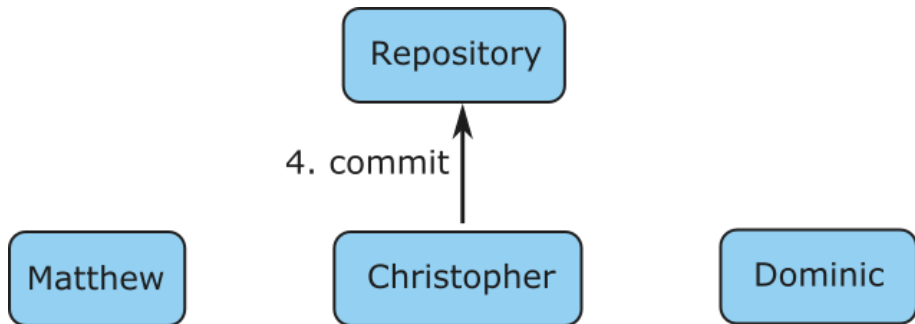
# SVN : processus



# SVN : processus

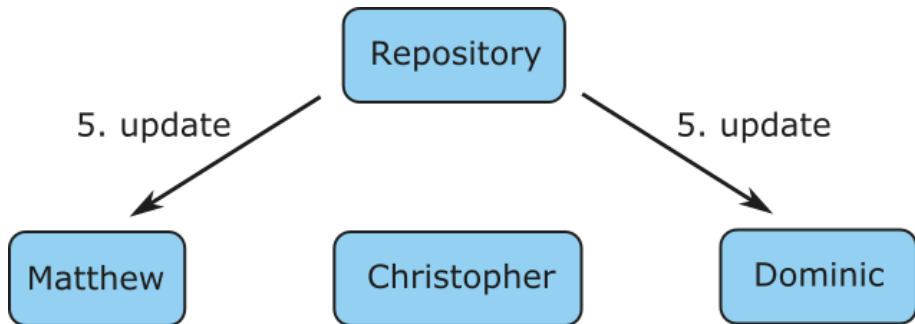


# SVN : processus

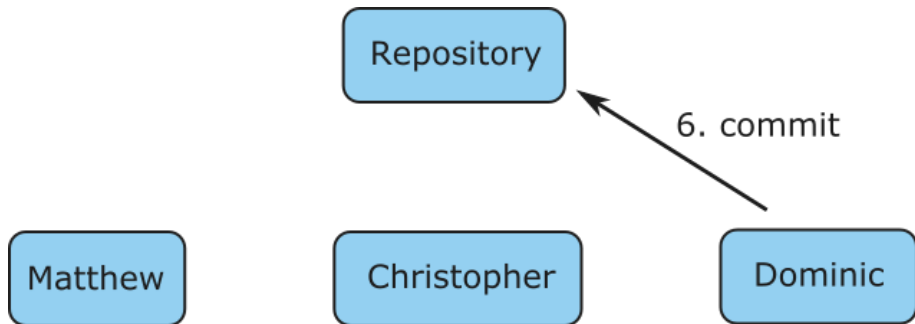




# SVN : processus



# SVN : processus



# SVN : exemple

démon

```
svnadmin create mondepot
```

```
svn import projet1 file:///<path>/mondepot/projet1
```

```
svn checkout file:///<path>/mondepot copie1/
```

Avant toute chose : `svn up`

Après modification du fichier f :

```
svn commit -m "j'ai modifié le fichier f"
```

Après création d'un fichier ff :

```
svn add ff
```

```
svn commit -m "j'ai ajouté ff"
```

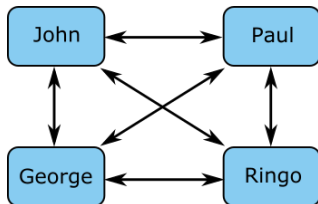
Pour supprimer le fichier ff :

```
svn rm ff
```

```
svn commit -m "j'ai supprimé ff"
```

Attention au répertoire dans lequel on exécute la commande SVN !

# Modèle distribué



- Plus de dépôt centralisé
- Chaque développeur a son propre dépôt avec ses branches privées
- Utilisation d'un réseau de dépôts équivalents: approche pair-à-pair
- Facilité d'échange des données entre dépôts par opérations push/pull
- Simplification de la fusion de branches: historique des fusions
- Influence sur la philosophie de développement : plus de liberté, mais risque de dispersion

# Modèle distribué

## Avantages

- Pas dépendant de la joignabilité d'un seul dépôt central
- Utilisable hors connexion

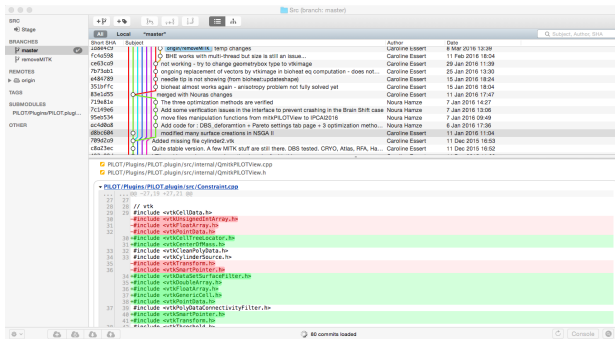
## Inconvénients

- Clonage lent car on copie tout l'historique
- Mise à jour à la version dispo des clients connectés
  - pas toujours la dernière !
  - solution : créer un dépôt principal qui héberge la toute dernière version (recentralisation)

# Modèle distribué

## Interfaces

- Visualisation des branches
- Diff graphique
- Historique des révisions avec date et commentaire



# Git : commandes principales

## Format général (`git <commande> [options]`)

- `init` : initialisation d'un dépôt git dans un rép existant
- `clone [url]` : récupérer localement une copie d'un projet distant existant (*équivalent du checkout de svn*)
- `add` : faire suivre un fichier par git / indexer un fichier comme "à envoyer" au prochain commit
- `commit -m "commentaire"` : valider les modifications dans les fichiers indexés
- `commit -a -m "commentaire"` : add + commit

# GIT : commandes principales

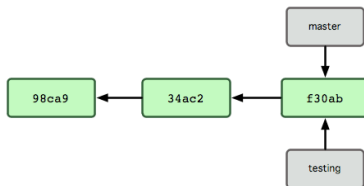
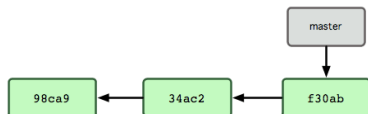
- **status** : connaître la branche courante et l'état des fichiers :
  - untracked : non suivi dans git
  - unmodified : suivi mais non modifié
  - modified : suivi et modifié
  - staged : suivi, modifié, et inclus dans la liste
- **diff** : montrer les lignes modifiées
- **rm** : indexer un fichier comme "à supprimer" au prochain commit
- **mv** : déplacer des fichiers
- **log** : afficher l'historique des révisions



# GIT : commandes principales

## Notion de branches

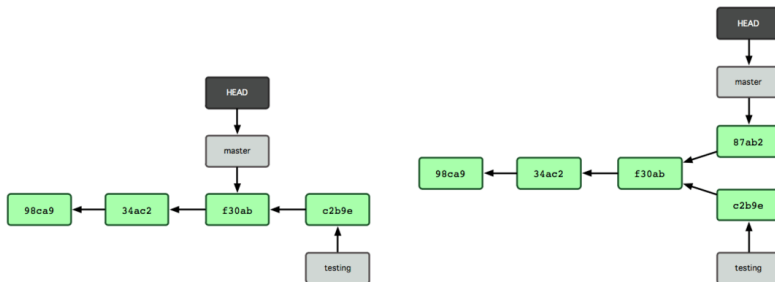
- Pointeur mobile léger vers un commit
- Branche par défaut = master
- `branch [nom-de-branche]` : crée une nouvelle branche, mais reste positionné sur la branche courante (ex. `git branch testing`)



# GIT : commandes principales

## Notion de branches

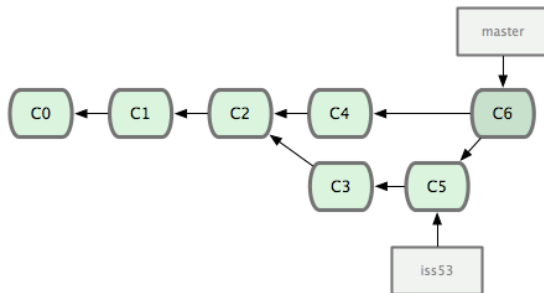
- `checkout [nom-de-branche]` : bascule vers une branche
- si commit sur la branche : le pointeur de la branche avance, pas le master (Fig. gauche)
- si autre commit sur le master : divergence des branches (Fig. droite)



# GIT : commandes principales

## Notion de branches

- `merge [nom-de-branche]` : fusion de branches (attention aux conflits !)
- `branch -d [nom-de-branche]` : supprime une branche (un pointeur de branche)



# **Git : commandes principales**

## **Travail avec un dépôt distant**

- `fetch` : récupère les données d'un dépôt distant qu'on ne possède pas déjà (sans fusion)
- `pull` : récupère les données d'un dépôt distant qu'on ne possède pas déjà (avec fusion)
- `push [nom-distant] [nom-de-branche]` : envoyer des données vers un serveur distant (ex: `git push origin master`)
  - **push pas toujours possible : pull request**
  - **il faut avoir fusionné les dernières modifs avant de faire un push ou pull request !**
- `remote show [nom-distant]` : inspecter un dépôt existant

## **Autres**

Etiquetage, etc., voir <https://git-scm.com/book/fr/v2>

# GIT : exemple

## démo

```
git init mondepot
git status
git add p1_f1
git add p1_f2
git status
git commit -m "ajout de deux fichiers"
git status
```

Sur une machine distante : `git clone [url]`

Sur le dépôt d'origine, après modification du fichier p1\_f1 :

```
git commit -a -m "j'ai modifié le fichier p1_f1"
```

Sur la machine distante, vérification des modifs : `git fetch`

Fusion : `git pull`

```
git status
```

Sur la machine distante : après modification du fichier p1\_f1 :

`git push` fait un message d'erreur

Sur le dépôt d'origine : `git pull [url]`

## Démon avec GitHub dépôt essert/TechDevDemo

The screenshot shows the GitHub interface for the repository `essert/TechDevDemo`. At the top, there's a search bar and navigation links for Pull requests, Issues, and Gist. Below the repository name, there are statistics: 1 Unwatch, 0 Stars, and 0 Forks. The main navigation bar includes links for Code, Issues (0), Pull requests (0), Projects (0), Wiki, Pulse, Graphs, and Settings. The repository description is "Demo project for Tech Dev students — Edit". Below this, there are statistics: 2 commits, 1 branch, 0 releases, and 1 contributor. A row of buttons includes "Branch: master", "New pull request", "Create new file", "Upload files", "Find file", and "Clone or download". A table lists the repository's files and their commit history:

File	Commit	Time
essert essai	Latest commit ac89278	23 minutes ago
README.md	Initial commit	25 minutes ago
toto.txt	essai	23 minutes ago

Below the table, the content of the selected `README.md` file is shown. It features the title **TechDevDemo** and the subtitle "Demo project for Tech Dev students".

- 1 GNU Debugger
- 2 Vérificateurs de la mémoire
- 3 Gestion de versions
- 4 Environnements de Développement Intégrés**
- 5 Documentation
- 6 Licence Logiciel
- 7 Archivage

# IDE

- IDE = Integrated Development Environment
- Logiciel qui gère des “projets”.
- Certains IDE sont dédiés à un langage de programmation donné, d'autres gèrent plusieurs langages (versions différentes, ou avec des greffons).
- IDE les plus connus :
  - Visual Studio (Windows)
  - Eclipse (Linux/Windows/MacOs)
  - Qt Creator (Linux/Windows/MacOs)
  - KDevelop (Linux)



Un IDE est un logiciel qui intègre différents outils qui inter-opèrent.

## Il contient toujours :

- Un éditeur de code.
- Des aides à la programmation : complétion, coloration syntaxique, indentation automatique, création de fichiers, etc.
- Une compilation automatisée (gestion des dépendances et des bibliothèques, construction du makefile, etc.).

## Il contient parfois :

- Un gestionnaire de versions
- Un debugger avec interface
- Un analyseur de mémoire

# Table des matières

- 1 GNU Debugger
- 2 Vérificateurs de la mémoire
- 3 Gestion de versions
- 4 Environnements de Développement Intégrés
- 5 Documentation**
- 6 Licence Logiciel
- 7 Archivage

# Documentation

- Outils de génération (semi-) automatique.
- Entrée : commentaires dans le code source.
- Sortie : documentation en texte, html, pdf, etc.
- Documentation destinée aux programmeurs :  
API = Application Programming Interface  
= Interface de Programmation.
- Exemples : Doxygen, JavaDoc.
- Intérêts :
  - écriture en même temps que le code,
  - mise à jour,
  - mise en page.

- Supporte plusieurs langages de programmation, dont :  
C, C++, Java, Python.
- Commentaires de code particuliers contenant des commandes.
- `doxygen -g` génère un fichier de configuration `Doxyfile`.
- `doxygen` génère la documentation.
- Quelques flags utiles :
  - `OPTIMIZE_OUTPUT_FOR_C`
  - `GENERATE_HTML`
  - `GENERATE_LATEX`
  - `OUTPUT_LANGUAGE`
  - `RECURSIVE`
- Pour débiter : tutoriel sur moodle.

- 1 GNU Debugger
- 2 Vérificateurs de la mémoire
- 3 Gestion de versions
- 4 Environnements de Développement Intégrés
- 5 Documentation
- 6 Licence Logiciel**
- 7 Archivage

# Logiciel libre

- Il s'oppose au logiciel propriétaire, où :
  - La reproduction, la distribution, l'ouverture et la modification du code source sont interdits.
  - L'exécution est limitée par la licence.
- On réfère au logiciel libre (free software) par rapport à la *liberté*, non à la *gratuité* (freeware).
- Un logiciel libre implique les 4 libertés suivantes:
  - La liberté d'exécuter le programme, pour tous les usages.
  - La liberté d'étudier le fonctionnement du programme, et de l'adapter à vos besoins. Pour cela l'accès au code source est une condition requise.
  - La liberté de redistribuer des copies.
  - La liberté d'améliorer le programme et de publier vos améliorations, pour en faire profiter toute la communauté. Pour ceci l'accès au code source est une condition requise.

# Licences

- La licence impliquant les quatre libertés précédentes est la GNU General Public License (GNU-GPL).
- Plusieurs autres licences existent, la différence majeure entre celles-ci est la notion de copyleft si importante dans la GPL.
- Copyleft : obligation de donner au minimum les mêmes droits aux logiciels améliorés qu'au logiciel original.
- "Open source" n'a pas le même sens que "logiciel libre".

# Conditions juridiques

La **licence GPL**, qui est la licence libre la plus utilisée, permet d'utiliser le logiciel, de le modifier et de le commercialiser ou de commercialiser sa nouvelle version, mais sans en faire un logiciel propriétaire. On doit en effet :

- le vendre sous la même licence libre GPL,
- transmettre en même temps un exemplaire de la GPL,
- être prêt à divulguer ses codes-sources,
- on ne peut pas empêcher qu'un autre ne diffuse gratuitement le logiciel ou sa nouvelle version.

D'**autres licences** libres (ex. BSD) permettent de commercialiser une nouvelle version d'un logiciel libre en faisant un logiciel propriétaire.



# Exemples

- Distributions Linux : Ubuntu, Debian, Mandriva, Redhat, Suse.
- Environnements de bureau : KDE, GNOME, Xfce.
- Navigateurs web : Mozilla Firefox, Epiphany (gnome), Konqueror (kde).
- Suites office : OpenOffice, Gnome Office, Koffice.
- Éditeurs graphiques : Gimp, ImageMagick, CinePaint, RubyMagick.

- 1 GNU Debugger
- 2 Vérificateurs de la mémoire
- 3 Gestion de versions
- 4 Environnements de Développement Intégrés
- 5 Documentation
- 6 Licence Logiciel
- 7 Archivage**

# Archivage

- Une archive est un fichier destiné à la sauvegarde ou à la transmission.
- Nombreux formats d'archives : **zip**, **tar**, **rar**, formats dédiés (paquets linux, code java), etc.
- Un format = une extension de fichier.
- Les archiveurs (logiciels) gèrent un ou plusieurs formats.
- Chaque archiveur possède des fonctionnalités qui peuvent être :
  - Paquetage : regroupe plusieurs fichiers dans une seule archive.
  - Compression : réduit la taille de l'archive.
  - Cryptage : crypte les données.
  - Correction d'erreur : permet de corriger d'éventuelles erreurs de transmission.
  - Ajout de commandes : l'ouverture de l'archive provoque des actions (par ex. installation ou compilation).

# Archivage : la commande tar

- La commande `tar` gère le format tar (paquetage), optionnellement combiné avec le format gzip (compression).

- Création d'une archive :

```
tar -c <fichiers ou répertoires> -f <archive.tar>
```

- Extraction d'une archive :

```
tar -x -f <archive.tar>
```

- Options classiques :

`-v` (verbose) : affiche la liste détaillée des fichiers.

`-z` (gzip) : compresse l'archive.

- Exemple de paquetage + compression :

```
tar -cvz fich.c rep/ -f sauvegarde.tar.gz
```

```
tar -xvz -f sauvegarde.tar.gz
```