Travail à plusieurs :

14/09/2022

git clone nom_du_dépôt
- → peut être un cabmin d'accès local
- → ssh
- → https

} copie l'arborescence du dépôt

La branche principale copiée est nommé origin/master

git pull ajout des modifications + merge

git branch -r affiche les branches y compris celles distantes

git checkout -t -b newbranch origin/newbranch se déplacer

git fetch synchronise les dépôt sans apporter de modification aux commits

git pull --rebase est équivalent à git fetch
git rebase origin/master

git push envoi les changements

git push --tags pour envoyer écrentuellement les tags

Remote : Versions distantes d'un même dépôt

git remote add nom_url

set-url nouveau nom_url

git blame --file montre le ID de l'auteur et la date de modification pour chaque ligne du fichier

Submodule : ajout d'un dépôt git à un projet plus gros

git submodule add url path ajoute un submodule contenu à url au chemin d'accès path

git submodule init initialise un submodule

Utile pour ajouter des blocs logiciel disponibles sur git (ex: pybind 11)

Ce lien est matérialisé par un dossier  (avec une flèche) sur GitHub.

Utilisation des Forges

Forge : Gestion de logiciel comprenant au moins :
- → dépot Git
- → autorisation
- → tragage des issues

Les forges sont hiérarchique :

https:// nom-de-domaine / responsable_ou_groupe / projet

Un problème est numéroté avec un # (ex : #4) appelé issue (levée par un user)

↳ facile d'y faire référence dans les messages de commit

git commit -m "add test for user input, ref #4"

=> fermeture automatique de l'issue par la forge si mot clé fix dans le message

=> référencement automatique

=> en pratique celui qui répare l'issue propose une pull request (propose de merge sa modification effectué sur une branche annexe pour la tester) avant de fermer

Hiérarchie goal des branches : (suite)

On crée une branche develop et on travaille develop

Une fois que l'on a une version stable on l'met sur la branche main

Lorsque l'on se trouve sur la branche develop mais que l'on souhaite ajouter une p'tite modification alors on se place sur une branche features

↳ Lorsque l'on a une version satisfaisante on effectue un pull request pour l'ajouter au develop

On assemble toutes les briques de develop sur une branche intermédiaire de release

↳ Si l'assemblage est validée alors on place le tout dans la branche main

Astuce : Faire apparaître la hiérarchie dans le nom des branche : topic / something

Forks désigne une scission dans un projet (dans le développement logiciel)

ou essayer de garder les versions synchronisés (dans le cas de forge)
=> cas étudié ici

Sur GitHub > Fork > Renseigner le responsable > choix de garder uniquement ou non la branche principale. Cela créé un clone du projet initial

Après avoir réalisé une modification GitHub propose de contribuer au projet qui a été forked. On peut réaliser une pull request (merge request sur GitLab) au collaborateurs du projet. Ces derniers pourront approuver

ou non les modifications.

Travailler localement avec plusieurs forks

```
git clone --recursive   clone avec les submodule s'il y en a
cd example-adder
git remote add          créer une branche distante
    git pull gepetto master
    git push origin master
```

Raccourci SSH (sur Linux) Évite de taper la ligne entière

Création d'un fichier ssh/config

Retour sur le projet

Installation du projet :

/ racine Linux (équivalent de C:\ sur Windows)

```
├ bin/   adder
├ lib/   libexample_adder.so
└ includes/
```

Linux possède une variable d'environement $PATH auquel on ajoute le chemin d'accès au dossier /bin

Pour Python, il est spécifié de procéder ainsi

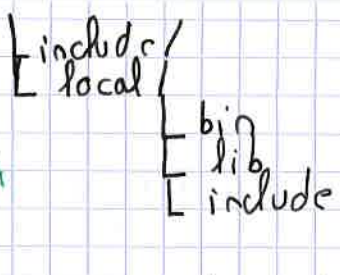
```
├ bin/
├ lib/
│   ├ libexample_adder.so
│   └ python3.10/
│       └ site-package/
│           └ Conception_orientee_objet/
│               ├ __init__.py
│               └ binary.mod
└ include
```

En pratique, tout n'est pas installé à la racine !

En fait / est un $PREFIX parmis d'autres. Ainsi, / ne contient que les fichiers important comme le /bin /bash

Il existe un autre $PREFIX principal appelé /usr/

```
├ bin/
└ lib/
```



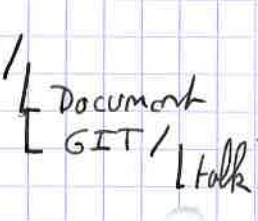
affiché avec echo $PATH

Ainsi, le $PATH = /bin : /usr/bin/usr/local/bin contient en fait plusieurs $PREFIX (séparation par :)

⚠ Tous les utilisateurs n'ont pas le droit en écriture ou en éxécution :

=> on peut choisir où installer comme par exemple /tmp/install
=> pas de droit par défaut de modifier /usr ou /.

L'utilisateur Linux a un répertoire qui l'appartient /home/gsaurel/



Création de l'environnement pour installer

```
mkdir /tmp/install
cd build
mkdir /tmp install/bin
mkdir /tmp/install /lib
cp libexample adder ...
```

tree /tmp/install affiche l'arborescence

export PATH = /tmp/install/bin : $PATH ajoute au $PATH le dossier où est installer l'éxécutable (ajoute à l'existant)

Deploiement du logiciel Dans CMakeLists.txt ajouter

```
install (TARGETS example-adder adder)
install (FILES include/conception-orientee-objet/example-adder.hpp
         DESTINATION include/conception-orientee-objet)
```

cmake --install build installe le projet dans /usr/local par défaut

Plutot que de forcer en temps qu'admin avec la commande sudo, on va donner une option à CMAKE

18 | cmake -B DCMAKE installe et créer automatiquement les dossiers

Packaging :

Solution 1 : Créer un .zip de tous le dossier résultant du build CMAKE

Solution 2 : Créer un paquet APT qui install par défaut au /usr / local

Afin de rendre le code réutilisable et exportable dans un autre projet :

On veut qu'un autre développeur puisse faire find_package (conception-orientee-objet) pour réutiliser les entêtes de fonction, etc..

Pour cela, on ajoute au CMakeLists.txt :

```
example-adder PUBLIC $<BUILD_INTERFACE:${CMAKE_SOURCE_DIR}/include>

install (TARGETS example-adder adder EXPORT conception_Orientee_objetTargets)
    install (
        EXPORT Conception-Orientee-objetTargets
        NAMESPACE Conception-Orientee-Objet::
        DESTINATION lib/cmake/Conception-Orientee-Objet)
    include (CMakePackageConfigHelpers)
```

Le préfixe d'installation contenant le chemin vers les fichiers d'intérêt est :

CMAKE_PREFIX_PATH = /tmp/install

Intégration continue : Créer un fichier au format yaml dans le dossier GitHub "workflow".

Dans l'onglet Actions sur GitHub : on voit les étapes de test de packaging afin de suivre l'intégration continue.

Lien Dynamique sur Linux : Permet d'indiquer l'emplacement des binaires avec

LD_LIBRARY_PATH = /tmp/install/lib

Ln fichier1 fichier2 effectue un lien "à la volé" dans l'environnement Linux.

Remarque : Pas besoin de cmake prefix path dans CMAKE_PREFIX_PATH si déjà dans la variable $PATH Derrière par D_LIBRARY_PATH=/tmp/install bien qui est contenue dans $PATH

Pour le faire depuis le fichier C_MAKELists.txt aussi :

Projet en Python

Ressources : Git nim655 / formes

Fichier formes.py

```
class Forme :
    color : str = "black"   # couleur par défaut
```

Dans un interpréteur

```
from formes import Form      # import de l'objet
f = Form()                   # instanciation
f.color                      # lecture d'un attribut
```

affiche 'black'

Fichier formes.py

```
class Forme :
    color : str = "black"
    def __str__(self) -> str:   # dunder qui change le comportement
        return f"forme indéfinie de couleur {self.color}"
```

définition d'une méthode

Dans l'interpréteur

```
f = Form()
print(f)
```

affiche forme indéfinie de couleur black

Fichier formes.py

```
class Form :
    """ classe de base pour les formes.
    >>> f = Form()
    >>> print(f)
    forme indéfinie de couleur noire
    """

if __name__ == "__main__":      # si appel du fichier alors instanciation automatique
    f = Forme()
    print(f)
```

print() appelle par défaut str

```
class rond(forme):
    a: float
    def __init__(self, a: float):          # constructeur
        self.a = a
    def __add__(self, other): return Rond(self.a + other.a)
```

r = Rond(10) par créer un rond et affecter 10 à a

```
class Quadrilatère (Forme):
    a: float
    b: float
    c: float
    d: float
```

Fonction qui retourne une forme du bon type:

Dans la classe Quadrilatère

```
def specialite(self):
    if self.a == self.b == self.c == self.d
        return "carré"
```

Prototype avec argument de taille variable

```
def show(*args, **kwargs)
```

Les arguments sont positionnels ou nommés en Python

*args : arguments positionnels capturés crée une liste des arguments

**kwargs : capture les arguments nommés créer un dictionnaire

```
if len(args) == 1:
    instance = Rond(args[0], **kwargs)
if len(args) == 4:
    instance = Quadrilatère(args[0], **kwargs)
print(instance)
```

Appel de la classe parente avec super()