Université Paris Diderot
Licence 3 Informatique

Conduite de Projet
Année 2015–2016

# Examen

## 27 juin 2016

**Instructions** Motivez vos réponses. Documents autorisés : 2 feuilles A4 recto-verso (4 pages). Tout dispositif électronique et/ou de communication est interdit. La durée de l'examen est de 3 heures.

**Exercice 1** [Gestion de versions, 4 points]

a) Quel est l'intérêt des outils `diff` et `patch` ? Quels sont les avantages de ces outils pour échanger des améliorations des fichiers *source*, par rapport à l'envoie des nouvelles versions des mêmes fichiers ? Y a-t-il des inconvénients ?

b) Expliquez la différence entre systèmes de gestion de version centralisés (p.ex., Subversion) et systèmes distribués (p.ex., Git).

**Exercice 2** [Preprocesseur, 3 points] Le fichier `hello.h` contient le code suivant :

```
#define PI 3

#ifdef DEBUG
#define MSG(s) printf("debug: %s\n", s);
#else
#define MSG(s)
#endif

void hello(void);
void bye() { printf("Bye!\n"); }
```

le fichier `hello.c` :

```
#include "hello.h"

void hello() {
    MSG("enter hello")
    printf("Hello, %d world!\n",
           PI);
    MSG("exit hello")
}
```

et le fichier `main.c` :

```
#include "hello.h"
#include "hello.c"

int main(void) {
    MSG("enter main")
    hello();
    bye();
    MSG("exit main")
}
```

a) Montrez le code C obtenu à la sortie du preprocesseur pour les fichiers `hello.c` et `main.c`.

b) Vous voulez maintenant habiliter l'affichage des messages de débogage. Comment pouvez vous le faire, en utilisant le preprocesseur ?

c) Que se passerait-t-il lors de la compilation de `main.c` ? Expliquez ce comportement. Si ce comportement est problematique, expliquez comment résoudre le problème en touchant seulement `hello.h`.

**Exercice 3** [*Make*, 4 points] Considérez le `Makefile` suivant :

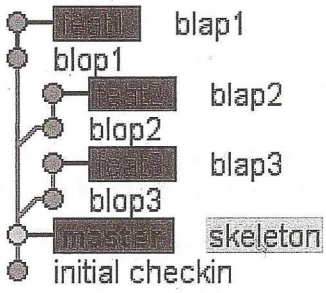
```
all: hello test
hello: main.o factorial.o hello.o
        g++ $^ -o hello
main.o: main.cpp
        g++ -c $<
factorial.o: factorial.cpp utils.hh
        g++ -c $<
hello.o: hello.cpp utils.hh
        g++ -c $<
test: hello
        /usr/local/bin/testrunner $<
clean:
        rm -rf *.o hello
```

a) Expliquez une règle à votre choix de ce `Makefile`, en montrant séparément son cible, prérequis, et commande.

b) Si aucun fichier `*.o` est présent (et si tous les autres fichiers mentionnés dans le `Makefile` existent), que se passera-t-il lors de l'exécution de `make test` ? Donnez la liste des commandes exécutées par `make`, dans un ordre valide.

c) Si, après une exécution complète de `make test`, vous modifiez `utils.hh` et ensuite exécutez `make hello`, que se passera-t-il ? Donnez à nouveau la liste des commandes exécutées par `make`, dans un ordre valide.

**Exercice 4** [Git, 4 points]
Écrivez une liste des commandes `git` qui produisent un dépôt Git dont l'historique ressemble le plus possible à celui montré en figure—où chaque noeud correspond à un *commit*, les textes sur la droite à des messages de *commit*, les étiquettes vertes à des noms des branches. Le contenu du dépôt en terme des fichiers n'est pas important (p.ex., vous pouvez utiliser un dépôt qui contient un seul fichier `toto.txt`), mais les *commits*, branches, et *merges* doivent correspondre à ceux en figure.




**Exercice 5** [Tests, 5 points]

a) Expliquez la différence entre tests unitaires, tests d'intégration, et tests de système.

b) Écrivez l'interface C minimale, dans le format d'un fichier `.h`, d'un type de données abstrait *ensemble d'entiers*. Proposez au moins 10 tests unitaires pour ce type de données, en donnant une courte description (= 1 phrase) pour chaque test. P.ex., « si on ajoute l'entier 42 à l'ensemble d'entiers vide, on obtient un ensemble d'entiers de cardinalité 1 ».

c) Montrez l'implémentation avec la librairie de test Check d'au moins 5 parmi les tests unitaires proposés.