



UFR IM²AG

**UNIVERSITÉ
Grenoble
Alpes**

Rapport de projet MÉTHODES ET OUTIL POUR LA CONCEPTION AVANCÉE

Puissance₄

GROUPE:6

MOUSS Adel
BELAID Mohamed
TAGUI Amine
ZITOUNI Hamza

Encadrants

LAURENCE PIERRE
LYDIE DU-BOUSQUET
LAURENT MOUNIER
DAVID MONNIAUX

Table des matières

Table des matières.....	2
Introduction :.....	3
Spécifications fonctionnelles et détaillées	3
Séance 1:	3
Séance 2:	4
Séance 3:	5
Séance 4 :	6
➤ Exercice 4 :	6
➤ Exercice 6 :	6
Séance 5:	7
➤ Exemple de programme contenant une erreur :	7
➤ Résultat de l'exercice 4:.....	9
Séance 6 :	9
➤ L'outil KLEE :	9
➤ L'outil valgrind :	11

Introduction :

Ce document a pour but de décrire le déroulement de notre projet de L'UE
MÉTHODES ET OUTILS POUR LA CONCEPTION AVANCÉE

Ce Projet Informatique porte sur un Logiciel déjà existant «Puissance_4» pour atteindre l'objectif fixé dans le cadre de la formation nous devons répondre à une succession définie par les points suivants:

- Introduction, lancement du projet (semaine 1).
- Modularité, maintenabilité, réutilisabilité (semaines 2 et 3).
- Qualité des tests, analyse de couverture (semaine 4).
- Tests pour l'analyse de vulnérabilité (semaine 5).
- Détection des défauts, correction de bugs (semaines 6 et 7).
- Résumé comparatif sur les méthodes pour le debug et pour l'analyse de vulnérabilité (semaine 8).
- Analyse de performances (semaines 9 et 10).

Ce rapport contient l'ensemble des éléments du projet:

- les spécifications plus détaillées qui en découlent.
- Nous décrirons le fonctionnement de notre projet dans son ensemble
- ainsi que les éléments qui prouvent le bon fonctionnement de celui-ci.

Spécifications fonctionnelles et détaillées

Séance 1:

Nous avons pris connaissance du programme, nous l'avons compilé à l'aide de la commande «gcc -Wall -Werror -g -o appli appli.c» pour corriger les erreurs mais aussi les warnings que nous avons observés.

Les warnings:

- warning: unused variable 'g'(supprimer la variable g)
- warning: control reaches end of non-void function
(ajouter un return 0 à la fin de la fonction)

Les erreurs de type «Segmentation fault»

- Problème de réallocation de mémoire pour la table de jeux
- Problème de réallocation de mémoire pour la sauvegarde d'une partie
les erreurs liées à la lecture et écriture dans les fichiers.

Séance 2:

Il convenait aujourd'hui principalement à diviser le programme en sous fichiers, chaque fichier contient des fonctions qui ont le même fonctionnement ou la même tâche :

- **appli.c** : Contient le programme principal (le « main »).
- **Initialisation.c** : Contient les fonctions d'allocation et d'initialisation.
- **Score.c** : Contient les fonctions qui calculent le score.
- **Check.c** : Contient les fonctions qui vérifient si la case choisie par le joueur est valide.
- **Iaplayer** : Contient les différents niveaux de jeux et les fonctions qui sont chargées à générer le déplacement de l'ordinateur si l'utilisateur choisi le mode solo.
- **GameBoard** : Contient les fonctions pour enregistrer le stage ou d'annuler/refaire un coup.

```
CC=gcc
CFLAGS=-Wall -Werror -g
ifdef N
    CPPFLAGS=-DN=$(N)
endif
SRCS=$(wildcard *.c)
OBJS=$(SRCS:.c=.o)
EXEC=appli

all:$(EXEC)

$(EXEC): $(OBJS)
    $(CC) $(OBJS) -o $(EXEC)

.PHONY:clean exec
clean:
    -rm $(OBJS)
exec: clean
    -rm $(EXEC)
```

-EXEC : contient le nom de l'exécutable.

- **Mode.c** : Contient les fonctions des différents modes de jeux.
- **Messages.c** : Contient la fonction d'affichage des scores.

La création du Makefile:

► Le Makefile a été fait pour qu'il soit le plus générique possible, pour cela, on avait besoin de déclarer les variables (cf. en dessous) avec des noms précis pour qu'elles soient reconnues par la règle implicite du Makefile.

Variables utilisées :

- CFLAGS : qui contient les Flags du langage « C ».
- CPPFLAGS : nécessaire pour la déclaration d'une macro.
- SRCS : pour récupérer les noms des fichiers « .c ».
- OBJS : qui sera utilisé pour la création des fichiers objets.

► L'option « -D » a été rajoutée pour l'unique raison de récupérer une valeur « N » au moment de l'appel du Makefile, à condition que cette option à bien été entrée dans la ligne de commande ainsi que la valeur voulue, et pour vérifier cela, on s'est servi de « ifdef ».

Séance 3:

Cette séance est portée sur la création des bibliothèques et le regroupement des fichiers dans des répertoires selon leurs extensions.

bin: appli.

Header:

appli.h
Initialisation.h
Score.h
Check.h
Iaplayer.h
GameBoard.h
Mode.h

SRC\OBJETS:

appli.c
Initialisation.c
Score.c
Check.c
Iaplayer.c
GameBoard.c
Mode.c

Messages.h

Messages.c

```
CC=gcc
CFLAGS=-Wall -Werror -IHeader -ILibMessages/Header
LDFLAGS=-L LibMessages/lib -l LibMessages
ifdef L
CPPFLAGS=-DL=$(L)
endif
ifdef C
CPPFLAGS=-DC=$(C)
endif
```

```
SRCS=$(wildcard src/*.c)
```

```
OBJS=$(SRCS:.c=.o)
EXEC=bin/appli
DOXYGENDIR = Doxygen
```

```
all:$(EXEC)
```

```
doxygen:
    doxygen ./Doxyfile
```

```
$(EXEC): $(OBJS)
    $(CC) $(OBJS) -o $(EXEC) $(LDFLAGS)
```

```
.PHONY:clean,doxygen
```

```
clean:
    -rm -rf $(OBJS) $(EXEC) $(DOXYGENDIR)
```

LIB :

LibMessages.so

Modification du Makefile :

► Ajout de l'outil de documentation automatique « doxygen » pour générer un fichier html.

► Ajout de la variable « LDFLAGS » pour inclure la bibliothèque créée.

Séance 4 :

Dans le but de corriger tout en développement le code comme il le faut, nous allons, dans cette séance, s'intéresser aux tests.

Nous allons utiliser les outils gcov et lcov pour être en capacité d'obtenir la couverture exacte de nos tests. En effet, pour que nos tests soient fiables, il faut qu'ils recouvrent idéalement la totalité des possibilités. Dans la pratique, en étant face à un nombre potentiellement infinis de possibilités, nous voudrions plutôt qu'ils couvrent un maximum de cas.

C'est pour cela que nous allons utiliser gcov et lcov, qui nous donnent tous les deux la couverture de notre code.

➤ Exercice 4 :

Augmenter la couverture de nos tests nous a poussé, d'une part, naturellement, à tester plus de possibilités, qui implique d'avoir un code plus fiable à l'issue de ces derniers. D'autre part, en s'ouvrant à des possibilités que nous n'avions pas prévu ou mal anticipé, nous pouvons revenir sur certains aspects du code et l'optimiser.

➤ Exercice 6 :

Tests en cours de construction.

Couverture des tests :

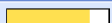
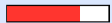


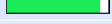


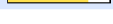
LCOV - code coverage report

Current view: [top level](#) - src

Test: rap.info

Date: 2019-02-08 11:54:02

	Hit	Total	Coverage
Lines:	496	681	72.8 %
Functions:	28	29	96.6 %

Filename	Line Coverage ↕	Functions ↕
Check.c	 82.1 % 23 / 28	80.0 % 4 / 5
GameBoard.c	 72.5 % 79 / 109	100.0 % 5 / 5
IPlayer.c	 61.9 % 78 / 126	100.0 % 4 / 4
Initialisation.c	 69.2 % 36 / 52	100.0 % 2 / 2
Messages.c	 91.5 % 65 / 71	100.0 % 1 / 1
Mode.c	 63.9 % 133 / 208	100.0 % 4 / 4
Score.c	 98.5 % 67 / 68	100.0 % 7 / 7
appli.c	 78.9 % 15 / 19	100.0 % 1 / 1

Generated by: [LCOV version 1.13](#)

Modification du Makefile :

```
CC=gcc
CFLAGS=-Wall -Werror -IHeader -IlibMessages/Header -fprofile-arcs -ftest-coverage
LDFLAGS=-L libMessages/lib -lmessages
ifdef L
CPPFLAGS=-DL=$(L)
endif
ifdef C
CPPFLAGS=-DC=$(C)
endif

SRCS=$(wildcard src/*.c)

OBS=$(SRCS:.c=.o)
EXEC=bin/appli
DOXYGENDIR = Doxygen
GCNO=src/*.gcno
GCDA=src/*.gcda

all:$(EXEC)

doxygen:
    doxygen ./Doxyfile

$(EXEC): $(OBS)
    $(CC) $(OBS) -o $(EXEC) -fprofile-arcs -ftest-coverage

.PHONY:clean,doxygen
clean:
    -rm -rf $(OBS) $(EXEC) $(DOXYGENDIR) $(GCNO) $(GCDA)
```

Séance 5:

➤ Exemple de programme contenant une erreur :

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    int t[10000];
    int i,c;
    FILE* f;
    f = fopen(argv[1],"r");
    fscanf(f,"%d",&c);
    for(i=0;i<c;i++){
        printf("---%d---\n",i);
        t[i]=i;
    }
    printf("t[%d]\n",t[i*2]);
    fclose(f);

    return 0;
}
```

Après avoir lancé AFL et laissé le FUZZER tourner, 2 crashes uniques ont été détectés (comme l'indique la capture d'écran en dessous).

```

american fuzzy lop 2.52b (bug)

process timing
  run time : 0 days, 0 hrs, 0 min, 51 sec
  last new path : 0 days, 0 hrs, 0 min, 51 sec
  last uniq crash : 0 days, 0 hrs, 0 min, 40 sec
  last uniq hang : none seen yet
cycle progress
  now processing : 1 (50.00%)
  paths timed out : 0 (0.00%)
stage progress
  now trying : havoc
  stage execs : 156/256 (60.94%)
  total execs : 11.3k
  exec speed : 197.8/sec
fuzzing strategy yields
  bit flips : 1/48, 0/46, 0/42
  byte flips : 0/6, 0/4, 0/0
  arithmetics : 0/336, 0/50, 0/0
  known ints : 0/36, 0/112, 0/0
  dictionary : 0/0, 0/0, 0/0
  havoc : 2/10.5k, 0/0
  trim : n/a, 0.00%

overall results
  cycles done : 17
  total paths : 2
  uniq crashes : 2
  uniq hangs : 0
map coverage
  map density : 0.00% / 0.01%
  count coverage : 1.00 bits/tuple
findings in depth
  favored paths : 2 (100.00%)
  new edges on : 2 (100.00%)
  total crashes : 23 (2 unique)
  total twouts : 0 (0 unique)
path geometry
  levels : 2
  pending : 0
  pend fav : 0
  own finds : 1
  imported : n/a
  stability : 100.00%

[Cpu000:102%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!

belaidm@im2ag-mandelbrot:~/L3/S6/moca/tp4/hello$

```

- 2 fichiers ont été créés par AFL pour provoquer ces crashes :

Fichier	Contenu
id:000000,sig:11,src:000001,op:havoc,rep:64	11111_€__111111111:11111E1011 1101111Q+111111111__1111
id:000001,sig:11,src:000001,op:havoc,rep:64	33333ÿ@ÿ3333333333__ÿÿ_3333ä

Après avoir lancé gdb avec les fichiers générés, la même erreur a été détectée pour les deux fichiers

```

Program received signal SIGSEGV, Segmentation fault.
0x000000000040067f in main (argc=2, argv=0x7fffffff158) at bug.c:11
11      t[i]=i;
(gdb)

```


➤ Résultat de l'exercice 4:

Pour un fuzzing de plus de 12 minutes, on obtient le résultat suivant :

american fuzzy lop 2.52b (appli)			
process timing		overall results	
run time : 0 days, 0 hrs, 12 min, 33 sec		cycles done : 0	
last new path : 0 days, 0 hrs, 0 min, 34 sec		total paths : 86	
last uniq crash : 0 days, 0 hrs, 1 min, 15 sec		uniq crashes : 15	
last uniq hang : 0 days, 0 hrs, 4 min, 22 sec		uniq hangs : 27	
cycle progress		map coverage	
now processing : 38* (44.19%)		map density : 0.72% / 1.06%	
paths timed out : 3 (3.49%)		count coverage : 1.75 bits/tuple	
stage progress		findings in depth	
now trying : bitflip 2/1		favored paths : 5 (5.81%)	
stage execs : 763/831 (91.82%)		new edges on : 14 (16.28%)	
total execs : 17.5k		total crashes : 4103 (15 unique)	
exec speed : 20.36/sec (slow!)		total tmouts : 10.4k (28 unique)	
fuzzing strategy yields		path geometry	
bit flips : 73/3392, 16/2557, 0/2551		levels : 3	
byte flips : 0/320, 0/317, 0/311		pending : 83	
arithmetics : 9/6048, 0/0, 0/0		pend fav : 5	
known ints : 0/0, 0/0, 0/0		own finds : 85	
dictionary : 0/0, 0/0, 0/0		imported : n/a	
havoc : 0/0, 0/0		stability : 100.00%	
trim : 2.97%/158, 0.00%			
[cpu001: 37%]			

Séance 6 :

➤ L'outil KLEE :

Le but de cette séance est de corriger toutes les erreurs concernant la mémoire, pour cela, l'outil KLEE a été utilisé.

Pour lancer KLEE sur notre projet (puissance 4), il a fallu modifier tout d'abord le Makefile pour inclure la bibliothèque contenant KLEE et aussi pour rajouter l'instruction qui génère un fichier "bitcode" qui, lui-même, sera exécuté après avec la commande "klee" pour afficher les erreurs détectées par cet outil.

Ce qui a été remarqué après l'exécution du fichier ".bc", de nombreuses erreurs ont été signalées concernant les fonctions de la librairie standard du "C", et c'est parce-que "Klee" n'arrive pas à trouver le chemin vers les fichiers où ces fonctions ont été déclarées ce qui explique les messages d'erreurs affichées, pour éviter cela, il est nécessaire de remplacer les "scanf" par des "lectures symboliques" (les "printf" peuvent être remplacés par des opérations qui ne fassent rien).

Les modifications apportées sur le Makefile :

```
CC=wllvm
CFLAGS=-Wall -g -IHeader -ILibMessages/Header -I $(KLEE)/include
LDFLAGS=-L LibMessages/lib
LDLIBS=-lMessages
ifdef L
CPPFLAGS=-DL=$(L)
endif
ifdef C
CPPFLAGS=-DC=$(C)
endif

SRCS=$(wildcard src/*.c)

OBJS=$(SRCS:.c=.o)
EXEC=bin/appli
DOXYGENDIR = Doxygen

all:$(EXEC) $(EXEC).bc

$(EXEC) : LibMessages.a

LibMessages.so: src/Messages.c
    $(CC) -fPIC -c src/Messages.c
    $(CC) -shared -o LibMessages/lib/LibMessages.so/Messages.o
LibMessages.a: src/Messages.o
    $(AR) cr $@ $+

doxygen:
    doxygen ./Doxyfile

$(EXEC): $(OBJS)
    $(CC) $(OBJS) -o $(EXEC)
    extract-bc $(EXEC)

.PHONY:clean,doxygen
clean:

-rm -rf $(OBJS) $(EXEC) $(DOXYGENDIR) $(EXEC).bc
```

➤ L'outil valgrind :

Pour cette partie, on n'avait pas besoin de modifier grand-chose, les seules modifications qui ont été effectuées ont été sur le Makefile, pour rajouter tout simplement le « -g » à la variable « CFLAGS » qui nous permettra de compiler notre programme avec l'option de débogage pour pouvoir l'exécuter ensuite avec « valgrind ».

Modification apportée sur le « Makefile » : l'ajout de l'option -g à la variable « CFLAGS » (cf ci-dessous).

```
CC=wllvm
CFLAGS=-Wall -g -IHeader -ILibMessages/Header -g
LDFLAGS=-L LibMessages/lib
LDLIBS=-lMessages
ifdef L
CPPFLAGS=-DL=$(L)
endif
ifdef C
CPPFLAGS=-DC=$(C)
endif

SRCS=$(wildcard src/*.c)

OBJS=$(SRCS:.c=.o)
EXEC=bin/appli
DOXYGENDIR = Doxygen

all:$(EXEC) $(EXEC).bc

$(EXEC) : LibMessages.a

LibMessages.so: src/Messages.c
    $(CC) -fPIC -c src/Messages.c
    $(CC) -shared -o LibMessages/lib/LibMessages.so/Messages.o
LibMessages.a: src/Messages.o
    $(AR) cr $@ $+

doxygen:
    doxygen ./Doxyfile

$(EXEC): $(OBJS)
    $(CC) $(OBJS) -o $(EXEC)

.PHONY:clean,doxygen
clean:

-rm -rf $(OBJS) $(EXEC) $(DOXYGENDIR)
```