



Guion de prácticas

Scripts apoyo NetBeans

Marzo 2020



Metodología de la Programación

DGIM

Curso 2019/2020

Índice

1. Descripción	5
1.1. Contenido	5
2. doConfig.sh	6
3. doDoxygen.sh	7
4. doUpdate.sh	8
5. doZipProject.sh	9
6. doTests.sh	11
6.1. Tests de salida básico	11
6.2. Tests de salida más complejos	12
6.3. Validando la salida de Valgrind	13
6.4. Validando programas que tienen salida a un fichero	14

1. Descripción

Este guión detalla las scripts que pueden instalarse en una carpeta dentro de un proyecto de NetBeans para realizar algunas tareas de gestión del proyecto de forma automática.

1.1. Contenido

- `scripts/doConfig.sh`
Script que contiene las variables principales de configuración del conjunto de scripts y que es invocada por todas las scripts al comienzo de su ejecución. Comprueba la estructura de carpetas del proyecto y, en caso de que así se indique con la variable `CREATE_FOLDERS` crea los que no existan.
- `scripts/doDoxygen.sh`
Ejecuta doxygen contra el fichero `.doxy` que haya colocado en la carpeta `doc/`, genera las salidas y llama a firefox para cargar la salida html.
- `scripts/doTests.sh`
Examina la carpeta `tests/` en busca de ficheros de autovalidación `.test`, los ejecuta y reporta el resultado. Opcionalmente permite pasar Valgrind en todas las ejecuciones de tests.
- `scripts/doUpdate.sh`
Actualiza las scripts desde la carpeta actual a todas las carpetas de NetBeans hermanas del proyecto actual. Se usa en el caso de que se hagan cambios en un set de scripts y estos se quieran propagar al resto de proyecto NetBeans.
- `scripts/doZipProject.sh`
Hace una copia de seguridad del proyecto. Para ello comprime el proyecto NetBeans para compartirlo en otros ordenadores, excluyendo algunas carpetas que no es necesario o conveniente compartir.

Todas ellas se pueden ejecutar desde dentro de Netbeans pulsando sobre ellas con el botón derecho y seleccionando **Run** o desde una terminal externa.

2. doConfig.sh

```
# Main
# Moves to root folder
CDRootFolder
# Reads root folder and its name
PROJECT_FOLDER=$(pwd)
PROJECT_NAME=$(basename $PROJECT_FOLDER)
# Create unexisting folders when required or not
CREATE_FOLDERS=YES
# Folder to store .cpp
CheckFolder src/ SRC_FOLDER
# Folder to store .h
CheckFolder include/ INCLUDE_FOLDER
# Folder to store .data
CheckFolder data/ DATA_FOLDER
# Folder to store the zip
CheckFolder zip/ ZIP_FOLDER
# Folder to store Doxygen's data
CheckFolder doc/ DOC_FOLDER
# Folder to store test data
CheckFolder tests/ TESTS_FOLDER
# Colors to be used in output of tests
RED='\033[0;31m'
GREEN='\033[0;32m'
GRAY='\033[1;30m'
WHITE='\033[1;37m'
# Use of memory leak detector
USE_VALGRIND=NO
# Forces the name of the ZIP. If left empty, the zip uses the name of the root folder
#ZIP_NAME=$PROJECT_NAME
ZIP_NAME="MPPractica"
```

3. doDoxygen.sh

```
#!/bin/bash
# Author: Luis Castillo Vidal L.Castillo@decsai.ugr.es
VERSION=1.0
# Load configuration & moves to the project root folder
if [ -d scripts ]
then
    source scripts/doConfig.sh
else
    source doConfig.sh
fi
# Runs doxygen
doxygen $DOC.FOLDER/*.doxy
# Display documentation
firefox $DOC.FOLDER/html/index.html &
```

4. doUpdate.sh

```
#!/bin/bash
# Author: Luis Castillo Vidal L.Castillo@decsai.ugr.es
VERSION=1.0
# Load configuration & moves to the project root folder
if [ -d scripts ]
then
    source scripts/doConfig.sh
else
    source doConfig.sh
fi
# Remember the current folder to come back to it at the end of the script
BASE=$(pwd)
echo "Exploring for other NetBeans project in the same subfolder..."
cd ..
# Copies the scripts in this folder into any other brother folder
for folder in $(ls)
do
    # Only copies scripts into Netbeans projects which also uses the folder scripts/
    if [ -d $folder/nbproject ] && [ -d $folder/scripts ] && [ ! $folder == $PROJECT_NAME ]
    then
        echo "Found Netbeans folder with scripts at $folder"
        cp $BASE/scripts/*.sh $folder/scripts
    fi
done
cd $BASE
```


5. doZipProject.sh

```
#!/bin/bash
# Author: Luis Castillo Vidal L.Castillo@decsai.ugr.es
VERSION=1.0
# Load configuration & moves to the project root folder
if [ -d scripts ]
then
    source scripts/doConfig.sh
else
    source doConfig.sh
fi

# Folders not to be included in the zip. Otherwise left empty
if [ $ZIP_NAME == $PROJECT_NAME ]
then
    EXCLUDED_FOLDERS="$PROJECT_NAME/nbproject/private/**\*_.*$PROJECT_NAME/dist/**\*_.*$PROJECT_NAME/build/**\*_.*$PROJECT_NAME/doc/html/**\*_.*$PROJECT_NAME/doc/latex/**\*"
else
    EXCLUDED_FOLDERS="nbproject/private/**\*_.*dist/**\*_.*build/**\*_.*doc/html/**\*_.*doc/latex/**\*"
fi

# Remove older copies
rm $ZIP_FOLDER/*.zip
# Zips the project. It either zips the whole NetBeans folder, or just the folders inside it
echo "Zipping project"
if [ $ZIP_NAME == $PROJECT_NAME ]
then
    cd ..
    eval "zip -r $PROJECT_NAME/$ZIP_FOLDER/$ZIP_NAME.zip $PROJECT_NAME/* -x $EXCLUDED_FOLDERS"
    cd -
else
    eval "zip -r $ZIP_FOLDER/$ZIP_NAME.zip * -x $EXCLUDED_FOLDERS"
fi
```

La variable `EXCLUDED_FOLDERS` define las carpetas que no se incluirán en la copia de seguridad como `nbproject/private/` (que almacena datos privados de configuración del proyecto), `dist/` y `build/` (que contienen binarios compilados) o `doc/html/` y `doc/latex/` que contienen documentación generada por doxygen.

La variable `ZIP_NAME` define el nombre que tendrá la copia de seguridad, la cual se almacenará en la carpeta `ZIP_FOLDER` definida en **doConfig.sh**. Si este nombre coincide con el nombre del proyecto entonces la copia de seguridad será de todo el proyecto desde su carpeta raíz. En otro caso la copia de seguridad será sólo de las carpetas internas del proyecto, sin incluir la carpeta raíz.

<pre> HelloWorld -- build -- Debug -- GNU-Linux -- HelloWorld.o -- HelloWorld.o.d -- main.o -- main.o.d -- data -- dist -- Debug -- GNU-Linux -- helloworld -- doc -- include -- Makefile -- nbproject -- configurations.xml -- Makefile-Debug.mk -- Makefile-impl.mk -- Makefile-Release.mk -- Makefile-variables.mk -- Package-Debug.bash -- Package-Release.bash -- private -- configurations.xml -- private.xml -- project.xml -- scripts -- doConfig.sh -- doDoxygen.sh -- doTests.sh -- doUpdate.sh -- doZipProject.sh -- src -- HelloWorld.cpp -- tests -- zip </pre>	<pre> . -- data -- doc -- include -- Makefile -- nbproject -- configurations.xml -- Makefile-Debug.mk -- Makefile-impl.mk -- Makefile-Release.mk -- Makefile-variables.mk -- Package-Debug.bash -- Package-Release.bash -- project.xml -- scripts -- doConfig.sh -- doDoxygen.sh -- doTests.sh -- doUpdate.sh -- doZipProject.sh -- src -- HelloWorld.cpp -- tests -- zip </pre>	<pre> HelloWorld -- data -- doc -- include -- Makefile -- nbproject -- configurations.xml -- Makefile-Debug.mk -- Makefile-impl.mk -- Makefile-Release.mk -- Makefile-variables.mk -- Package-Debug.bash -- Package-Release.bash -- project.xml -- scripts -- doConfig.sh -- doDoxygen.sh -- doTests.sh -- doUpdate.sh -- doZipProject.sh -- src -- HelloWorld.cpp -- tests -- zip </pre>
Carpetas originales	ZIP_NAME= MPP practica	ZIP_NAME= \$PROJECT_NAME
	MPP practica.zip	Helloworld.zip

6. doTests.sh

Esta es la script más compleja de todas y está orientada según la metodología “Test Driven Development”¹ en la cual, antes de iniciar el desarrollo se definen una serie de pruebas que debe cumplir el programa, las cuales, una vez satisfechas, podrían dar por validado el programa hasta un cierto grado, que depende de cómo de exhaustivo sea el conjunto de pruebas.

Esta script recorre la carpeta **tests/** o la que se haya definido en **do-Config.sh** y, por cada fichero **.test** encontrado, realiza una llamada al binario del proyecto y comprueba que la salida sea la correcta.

6.1. Tests de salida básico

El fichero test tiene este formato.

```
FICHERO: v_4inside.test
0 10 10 0
0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
-1 0

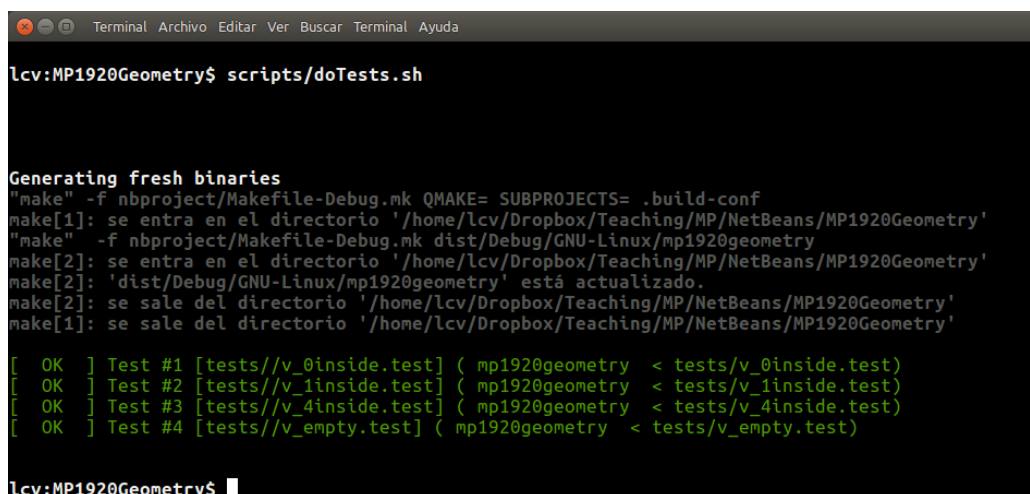
%%CALL < tests/v_4inside.test
%%OUTPUT
First rectangle is [(2,5) - (5,2)]
Type second rectangle:
Calculating intersection of: [(2,5) - (5,2)] and [(0,10) - (10,0)]
The intersection is: [(2,5) - (5,2)]
Reading points...(2,2) (3,3) (4,4) (5,5) fall within the intersection (4 total)
```

En primer lugar aparecen los datos que se le van a pasar al programa mediante redirección de la entrada. En verde aparece un código que indica cómo será la llamada que se ejecute para este test. Se puede ver que redirecciona la entrada desde este mismo fichero. Como la entrada de datos termina cuando se lee un número negativo, el programa terminará leyendo el último valor -1 0 y no entrará en esta zona de códigos. A continuación, en azul, aparece la descripción de la salida del programa cuando se ejecuta esta redirección de datos. Así, la script ejecuta el binario, le redirecciona los datos del fichero, captura la salida y comprueba que esta sea la misma.

La Figura 1 muestra la ejecución de la script **doTests.sh**, la cual encuentra cuatro ficheros de validación **.test**. Analiza cada uno de ellos, ejecuta la llamada indicada, captura la salida del programa y la compara con la contenida en el fichero **.test** para ver si coinciden. Si coinciden, el test

¹https://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas

se da por válido y se pasa al siguiente. Cuanto mejor diseñado y exhaustivo sea el conjunto de tests, más alta es la probabilidad de que el programa sea válido si pasa todos los tests.



```

lcv:MP1920Geometry$ scripts/doTests.sh

Generating fresh binaries
"make" -f nbproject/Makefile-Debug.mk QMAKE= SUBPROJECTS= .build-conf
make[1]: se entra en el directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Geometry'
"make" -f nbproject/Makefile-Debug.mk dist/Debug/GNU-Linux/mp1920geometry
make[2]: se entra en el directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Geometry'
make[2]: 'dist/Debug/GNU-Linux/mp1920geometry' está actualizado.
make[2]: se sale del directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Geometry'
make[1]: se sale del directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Geometry'

[ OK ] Test #1 [tests//v_0inside.test] ( mp1920geometry < tests/v_0inside.test)
[ OK ] Test #2 [tests//v_1inside.test] ( mp1920geometry < tests/v_1inside.test)
[ OK ] Test #3 [tests//v_4inside.test] ( mp1920geometry < tests/v_4inside.test)
[ OK ] Test #4 [tests//v_empty.test] ( mp1920geometry < tests/v_empty.test)

lcv:MP1920Geometry$

```

Figura 1: Ejecución con éxito del conjunto de tests. Cada test reporta el fichero .test en el que se basa, la llamada que se produce y el resultado del test. Los tests pasados se muestran de color verde en la terminal

La Figura 2.a) muestra una ejecución de **doTests.sh** en la que una de las pruebas ha fallado. La script lo señala y pregunta si se desea un informe detallado de cuál ha sido el fallo. Si la respuesta es y la script muestra un informe por cada test que haya fallado (Figura 2.b). En ese informe se muestra cuál ha sido la salida real obtenida por el programa al ejecutarse y en qué se diferencia de la salida que debería haber salido y que se encuentra descrita en el fichero de test asociado.

6.2. Tests de salida más complejos

El procedimiento anterior comprueba toda la salida del programa, carácter a carácter. Sin embargo, en programas en los que la salida es muy compleja o puede variar entre implementaciones, puede que sólo se quiera comprobar una parte de dicha salida, como por ejemplo, la salida de resultados finales. En ese caso, se recomienda mostrar la salida siempre con la misma función y forzar el programa a escribir en la pantalla los códigos `%%OUTPUT` de forma que la script pueda parsearla y comprobar sólo la salida desde que estos códigos aparecen en la pantalla en adelante. El resto de la salida del programa se ignoraría. En caso de que se deba validar que el programa detecta e informa de situaciones de error, estos mensajes de error también deberían estar marcados con los códigos `%%OUTPUT`. Este es el caso de las funciones `showResults(...)` que muestra de forma controlable y parseable las principales variables de salida de un programa y `errorBreak(...)` que muestra los principales errores que debería detectar el programa.

```

...
#define ERROR_ARGUMENTS 1
#define ERROR_OPEN 2
#define ERROR_DATA 3

void errorBreak(int errorcode, const string &errordata) {
    cerr << endl << "%%OUTPUT" << endl;
    switch(errorcode) {
        case ERROR_ARGUMENTS:
            cerr<<"Error_in_call.Please_use:\n_-l<language>_-r<randomnumber>_-i<inputfile >]"<<endl;
            break;
        case ERROR_OPEN:
            cerr<<"Error_opening_file_"<<errordata << endl;
            break;
        case ERROR_DATA:
            cerr<<"Data_error_in_file_"<<errordata << endl;
            break;
    }
    std::exit(1);
}

void showResults(const Language &l, int random, const Bag &b, const Player &p,
                const Movelist& original, const Movelist& legal,
                const Movelist& accepted, const Movelist& rejected) {
    cout << endl << "%%OUTPUT" << endl << "LANGUAGE:_"<<l.getLanguage()<< "ID:_" << random << endl;
    cout << "BAG_("<<b.size()<<"):_)" << toUTF(b.to_string()) << endl;
    cout << "PLAYER_("<<p.size() << "):_)" << toUTF(p.to_string());
    cout << endl << endl << "ORIGINAL_("<<original.size()<<"):_)"<<endl; original.print(cout);
    cout << endl << endl << "LEGAL_("<<legal.size()<<"):_)"<<endl; legal.print(cout);
    cout << endl << endl << "ACCEPTED_("<<accepted.size()<<")_.SCORE_"<<accepted.getScore()<< ":_)"<<endl;
    accepted.print(cout);
    cout << endl << endl << "REJECTED_("<<rejected.size()<<"):_)"<<endl; rejected.print(cout);
    cout << endl;
}

...
int main(int nargs, char * args[]) {
    ...
    if (arg>=nargs)
        errorBreak(ERROR_ARGUMENTS, "");
    ...
    ifile.open(ifilename);
    if (!ifile) {
        errorBreak(ERROR_OPEN, ifilename);
    }
    ...
    showResults(language, Id, bag, player,
                movements, legalmovements, acceptedmovements, rejectedmovements);
    return 0;
}

```

6.3. Validando la salida de Valgrind

En caso de necesitar validar la ejecución de un programa mediante Valgrind se puede hacer de dos formas distintas. La primera es poner la variable `USE_VALGRIND=YES` dentro de la script **doConfig.sh**. Esto hace que todos los tests se ejecuten con Valgrind. La segunda opción es marcar sólo algunos de los tests para ejecutarse con Valgrind. En ese caso basta con incluir los códigos `%%VALGRIND` dentro de aquellos ficheros **.test** que se quieran ejecutar con Valgrind.

```

FICHERO: EN_EU_2020_U.test
%%CALL -r 2020 -l EN -i data/EU_2020_UNSORTED.data
%%VALGRIND
%%OUTPUT
LANGUAGE: EN ID: 2020
BAG (37): AEAAMNAGDGYEHLIEJBSOWDWNQSPULEFOYEKNO
PLAYER (7): ACEHIU
ORIGINAL (49):
LEGAL (29):
ACCEPTED (13) SCORE 90:
REJECTED (16):

```

En este caso, un test falla cuando la salida no coincide o cuando Valgrind detecta errores de memoria (ver Figura 3).

6.4. Validando programas que tienen salida a un fichero

Hay programas cuya salida no es en pantalla, o al menos no sólo es en pantalla, sino que además pueden dar la salida en un fichero y es necesario validar que ese fichero está bien definido. Para ello se utiliza la marca `%%%FROM_FILE <path-to-file>`

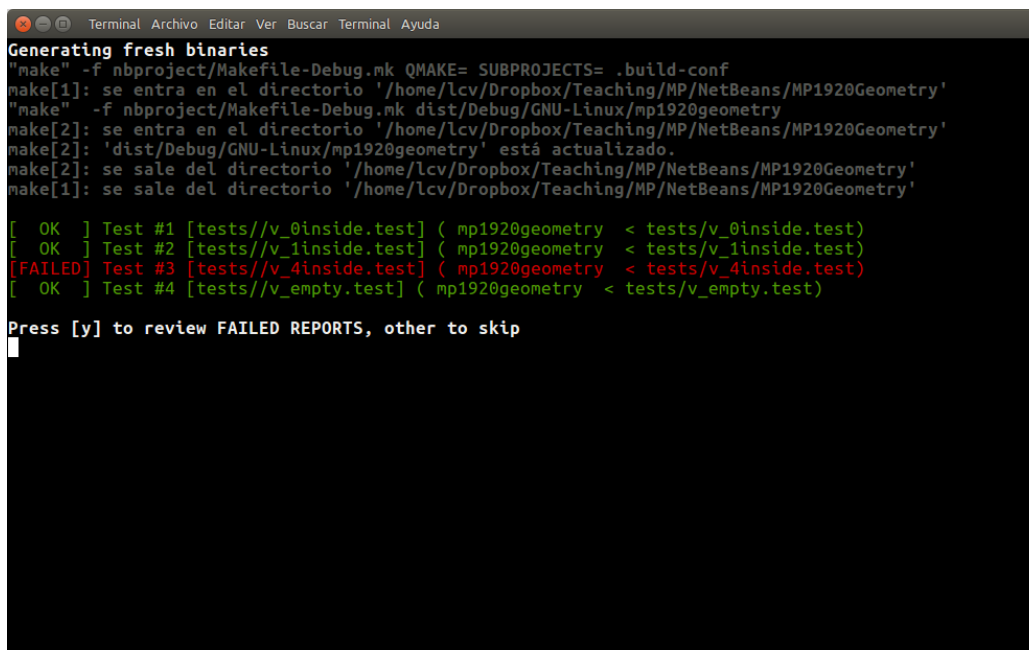
[illegible]

El ejemplo anterior invoca al programa (%%%CALL) con una serie de parámetros

```
-l ES -r 2020 -w 15 -h 15
lee los datos de entrada de un fichero
-p data/ES_2020_155_A.data
y escribe la salida en otro fichero
-save data/ES_2020_98.match
```

En este caso, indicamos en el fichero de validación que la salida a validar no es la salida por pantalla, sino la salida del fichero que aparece en la llamada

%%%FROMFILE data/ES_2020_98.match
por lo que la salida a comprobar (%%%OUTPUT) no es la salida por pantalla,
sino el contenido de ese fichero.



```

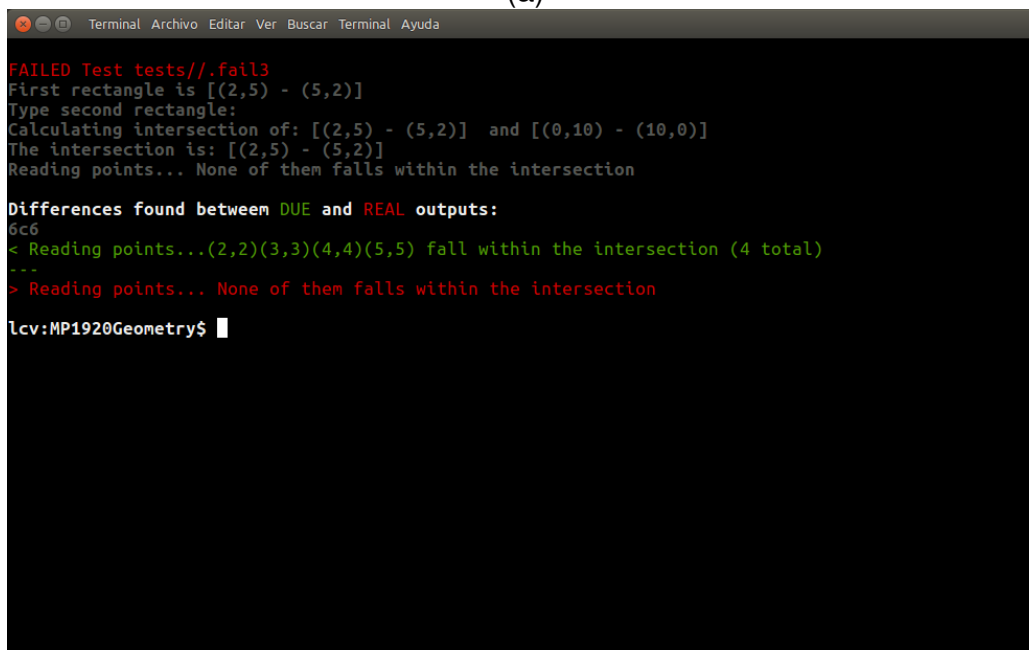
Terminal Archivo Editar Ver Buscar Terminal Ayuda
Generating fresh binaries
"make" -f nbproject/Makefile-Debug.mk QMAKE= SUBPROJECTS= .build-conf
make[1]: se entra en el directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Geometry'
"make" -f nbproject/Makefile-Debug.mk dist/Debug/GNU-Linux/mp1920geometry
make[2]: se entra en el directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Geometry'
make[2]: 'dist/Debug/GNU-Linux/mp1920geometry' está actualizado.
make[2]: se sale del directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Geometry'
make[1]: se sale del directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Geometry'

[ OK ] Test #1 [tests//v_0inside.test] ( mp1920geometry < tests/v_0inside.test)
[ OK ] Test #2 [tests//v_1inside.test] ( mp1920geometry < tests/v_1inside.test)
[ FAILED ] Test #3 [tests//v_4inside.test] ( mp1920geometry < tests/v_4inside.test)
[ OK ] Test #4 [tests//v_empty.test] ( mp1920geometry < tests/v_empty.test)

Press [y] to review FAILED REPORTS, other to skip

```

(a)



```

Terminal Archivo Editar Ver Buscar Terminal Ayuda
FAILED Test tests//v_4inside.test
First rectangle is [(2,5) - (5,2)]
Type second rectangle:
Calculating intersection of: [(2,5) - (5,2)] and [(0,10) - (10,0)]
The intersection is: [(2,5) - (5,2)]
Reading points... None of them falls within the intersection

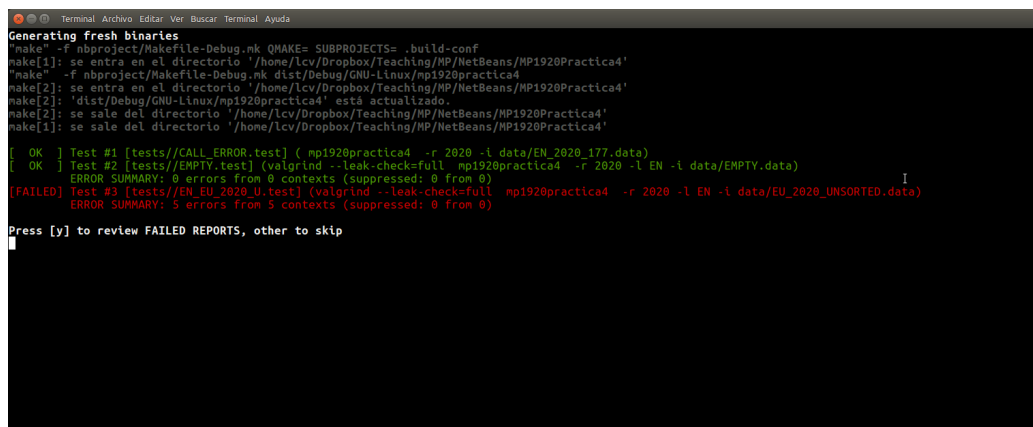
Differences found between DUE and REAL outputs:
6c6
< Reading points...(2,2)(3,3)(4,4)(5,5) fall within the intersection (4 total)
---
> Reading points... None of them falls within the intersection

lcv:MP1920Geometry$ 

```

(b)

Figura 2: Fallo en la ejecución del test número 3 contenido en el fichero *v_4inside.test*. a) El test que ha fallado aparece en color rojo. b) El reporte de los tests fallados muestra en verde cómo debería haber sido la salida y en rojo como ha sido en realidad



```

Generating fresh binaries
"make" -f nbproject/Makefile-Debug.mk QMAKE= SUBPROJECTS= .build-conf
make[1]: se entra en el directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Practica4'
"make" -f nbproject/Makefile-Debug.mk dist/Debug/GNU-Linux/mp1920practica4
make[2]: se entra en el directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Practica4'
make[2]: 'dist/Debug/GNU-Linux/mp1920practica4' está actualizado.
make[2]: se sale del directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Practica4'
make[1]: se sale del directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/MP1920Practica4'

[ OK ] Test #1 [tests//CALL_ERROR.test] ( mp1920practica4 -r 2020 -l data/EN_2020_177.data)
[ OK ] Test #2 [tests//EMPTY.test] (valgrind --leak-check=full mp1920practica4 -r 2020 -l EN -l data/EMPTY.data)
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[FAILED] Test #3 [tests//EN_EU_2020_U.test] (valgrind --leak-check=full mp1920practica4 -r 2020 -l EN -l data/EU_2020_UNSORTED.data)
ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)

Press [y] to review FAILED REPORTS, other to skip

```

Figura 3: Test#1 se ejecuta SIN Valgrind y la salida es correcta. Test#2 se ejecuta con Valgrind, la salida es correcta y la gestión de memoria correcta. Test#3 se ejecuta con Valgrind y Valgrind reporta 5 errores. El informe de fallos dará información extensiva sobre las salidas de los tests que han fallado, incluyendo la salida de Valgrind