Guion de prácticas

Test Driven Development

Google Test & MPTest

Febrero de 2021

Metodología de la Programación

DGIM-GII-GADE

Curso 2020/2021

Índice

1. Introducción

El El objetivo de estea documento es hacer una breve introducción a la metodología TDD (*Test-Driven Development*) y a su puesta en marcha usando una suite de testeo conocida (Google Test) a la que se le han añadido algunas funciones específicas de esta asignatura, con el doble objetivo de introducir esta metodología y que la experiencia de aprendizaje sea lo más estándar posible para los alumnos.

1.1. Test Driven Development

TDD es una metodología de desarrollo de aplicaciones que se basa en escribir, en primer lugar, los tests que debe de pasar correctamente el software que se va a crear y, posteriormente, crear ese software con el objetivo fundamental de que pase todos los tests para, posteriormente, adaptarlo a un diseño y/o arquitectura específica. Eso sí, sin dejar de pasar nunca todos los tests. Quizás los dos objetivos fundamentales de TDD sean buscar el diseño más sencillo posible, evitando toda complejidad innecesaria (KISS princple[1]) y, por otro lado, proporcionar confianza en el desarrollo de software.

Hay varios tipos de tests, que varían según la aplicación que se esté desarrollando, entre los que podemos distinguir los más básicos:

- Tests Unitarios[2]. Cuyo objetivo es comprobar que cada unidad funcional mínima que se construya, ya sea una función o un método, cumple una serie de buenas propiedades. Así, podemos considerar que la implementación de una clase es correcta si todos sus métodos han sido probados individualmente, lo cual ofrece una base de trabajo garantizada desde la que desarrollar nuevas clases o construir el programa completo.
- Tests de Integración [3]. Cuyo objetivo es comprobar que la integración de todas las clases construidas, la entrada y salida de datos, sigue siendo correcta, partiendo de la base de que las clases se han probado correctas con tests unitarios.

[1]KISS = Keep It Simple Stupid, (Abrir en navegador →)
[2]Unit Testing (Abrir en navegador →)
[3]Integration Testing (Abrir en navegador →)

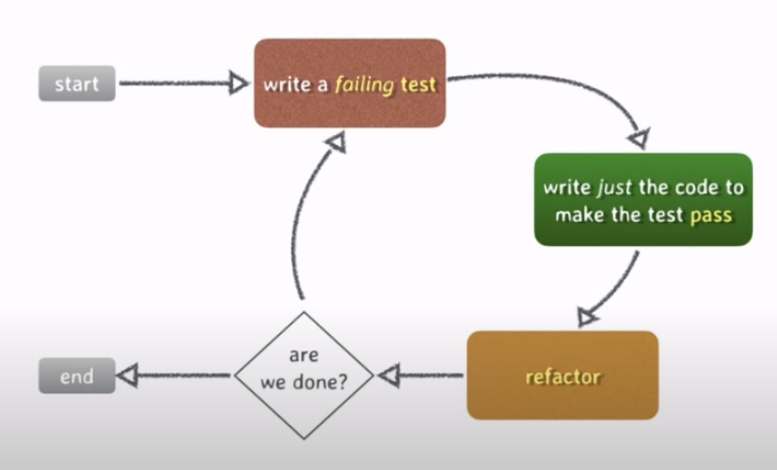


Figura 1: Ciclo de vida de TDD. Phil Nash, CppCon The C++ Conference, 2020. (Abrir en navegador →)

Los pasos a seguir en la metodología TDD, que no son muy diferentes del proceso tradicional, sólo que están más ordenados (ver Figura 1), son los siguientes (en negrilla los que debe de hacer el alumno, en tipografía normal los pasos que aporta el profesor).

1. Escribir el diseño de la API de la clase, sólo los métodos.
2. Escribir los tests unitarios y de integración.
3. **Escribir el código de los métodos hasta que pasen todos los tests unitarios, sea como sea.**
4. **Cuando pasen todos los tests, intentar refactorizar el código para mejorarlo, en cuyo caso habría que volver al paso 3 para testear los cambios hechos.**
5. **Escribir el main() para que pase los tests de integración, sea como sea.**
6. **Es posible que haya que refactorizar alguna clase previa. Si es así, volver al paso 3.**
7. **Si ha pasado todos los tests de integración, intentar refactorizar el código para mejorarlo y volver al paso 5.**
8. **Compilar la versión definitiva.**

```
class Number {
private:
int data;
public:

Number() { data = 0; }
int get() const { return data; }
int increase() {data++;}
void add(Number n) {data += n.get();}
bool isEven() {return data %2 == 0;}
};
```

Figura 2: Fichero `Number.h`

2. TDD con GoogleTest

Esta sección describe cómo se puede aplicar esta metodología, en el caso concreto de la asignatura Metodología de la Programación. Para aplicar TDD a un proyecto de desarrollo de software necesitamos dos cosas:

1. En primer lugar, un entorno de testeo, de los que hay muchas opciones, y suelen consistir en una librería de funciones preparadas para realizar los tests sobre nuestro proyecto. En este caso se va a usar la suite de testeo **GoogleTest** desarrollada por Google para testear aplicaciones escritas en C++[4] y se ha ampliado para cubrir también unos sencillos tests que no cubre GoogleTest (extensión **MPTest**).
2. Un entorno de desarrollo en el que integrar el entorno de testeo con nuestro proyecto. De nuevo hay muchas opciones, pero se ha elegido **Apache NetBeans 12** [5].

2.1. Principales herramientas de GoogleTest: definición de tests unitarios

GoogleTest es una suite de testeo muy potente y con muchísimas funcionalidades que se pueden consultar en si Git o en las secciones **??** y sucesivas, extraídas, precisamente, del Git de Googletest, las más básicas de las cuales se describen aquí.

Supóngase la siguiente clase (Figura 2, cuya implementación se ha incluído para hacerla autoexplicativa y más fácil de introducir los tests unitarios.

La programación de tests unitarios pretende validar el funcionamiento de cada método o función sin necesidad de haber programado aún la función `main()` y suelen programarse mediante macros que emulan llamadas a funciones. En este caso se usa la macro `TEST(C,T)` con dos parámetros, el primero es un identificador que representa al conjunto de tests `C` y el segundo, al test unitario en particular `T` dentro de ese conjunto de tests. Normalmente cada clase define un conjunto de tests, y cada método al menos un test individual. Estos tests (Figura 3) se escriben en ficheros `.cpp` que forman parte del proyecto, pero no se almacenan en la carpeta `src` sino en la carpeta `tests`.

[4]GoogleTest (Abrir en navegador →)
[5]Apache NetBeans 12 (Abrir en navegador →)

```
#include <gtest/gtest.h>
#include "Number.h"

using namespace std;

TEST(C, T) {

//cuerpo del test

}
```

Figura 3: Fichero de tests, que podría llamarse `Unit_Number.cpp` y que estaría ubicado en la carpeta `tests` y forma parte del proyecto

El cuerpo del test se puede programar como una función más en C++.

```
TEST(Number, Number_increase) {
Number a;        // Se inicializa a 0
a.increase();    // Se incrementa en 1
}
```

Con el principal añadido que se realizan aserciones para comprobar propiedades que el método debe cumplir y estas aserciones son macros que se definen en GoogleTest, como esta, a continuación que comprueba la igualdad de dos valores con `ASSERT_EQ`

```
TEST(Number, Number_increase) {
Number a;        // Se inicializa a 0
a.increase();    // Se incrementa en 1

ASSERT_EQ(a.get(),1);
}
```

Si ambos valores coinciden, se da el test por superado (aparece en verde) y, si fuese el único test, se daría por pasada la fase de test completa. En la Figura siguiente aparece el inicio y el final de toda la fase de test con la marca

`[========]` `Inicio y final la fase de testeo`

Y cada conjunto o agrupación de tests, lo que se conoce como "test suite", aparece marcado al inicio y al final con la marca

`[--------]` `Inicio y final de un grupo de tests (suite)`

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from Number
[ RUN      ] Number.Number_increase
[       OK ] Number.Number_increase (0 ms)
[----------] 1 test from Number (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

en otro caso, el test falla (aparece en rojo) y GoogleTest nos explica por qué: "Ha salido 3, cuando esperaba que hubiese salido 1". En este caso, el fallo de un único test, hace que toda la fase de tests falle también.

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from Number
[ RUN      ] Number.Number_increase
src/main.cpp:48: Failure
Expected equality of these values:
  a.get()
    Which is: 3
  1
[  FAILED  ] Number.Number_increase (0 ms)
[----------] 1 test from Number (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] Number.Number_increase

 1 FAILED TEST

RUN FINISHED; exit value 1; real time: 0ms; user: 0ms; system: 0ms
```

En estos casos, se puede añadir un texto explicativo, como si fuese `cout` junto a la aserción.

```
TEST(Number, Number_increase) {
Number a;          // Se inicializa a 0
a.increase();      // Se incrementa en 1
ASSERT_EQ(a.get(),1)<<"Si incrementamos una instancia recién creada de Number, el
    resultado debe ser 1" << endl;
}
```

```
[----------] Global test environment set-up.
[----------] 1 test from Number
[ RUN      ] Number.Number_increase
src/main.cpp:48: Failure
Expected equality of these values:
  a.get()
    Which is: 3
  1
Si incrementamos una instancia recién creada de Number, el resultado debe ser 1

[  FAILED  ] Number.Number_increase (0 ms)
[----------] 1 test from Number (0 ms total)
```

El incumplimiento de esta aserción daría por no pasada la fase de test y obligaría a escribir o refactorizar el código para que lo pudiese pasar.

Además de `ASSERT_EQ` existen muchas más aserciones, dependiendo de la propiedad que se quiera comprobar, aparecen todas descritas en las Secciones ?? y ?? y aparecen resumidas en el Cuadro 1.

`ASSERT_EQ(expr1, expr2)`	Las dos expresiones deben ser iguales
`ASSERT_NE(expr1, expr2)`	Las dos expresiones deben ser diferentes
`ASSERT_TRUE(expr1)`	La expresión debe ser verdadera
`ASSERT_FALSE(expr1)`	La expresión debe ser falsa
`ASSERT_STREQ(char*1, char*2)`	Las dos cadenas deben ser iguales
`ASSERT_STNE(char*1, char*2)`	Las dos cadenas deben ser iguales
`ASSERT_DEATH(expr1, message)`	Al evaluar la expresión, debe saltar un `assert`
`ASSERT_EQ(expr1, expr2)`	Las dos expresiones deben ser iguales

Cuadro 1: Algunas de las asserciones más frecuentes de GoogleTest

Aunque todo esto se puede hacer sin definir un `main()`, sí que es necesario inicializar la librería de tests y ordenar la ejecución de los tests, por lo que el main realmente quedaría así:

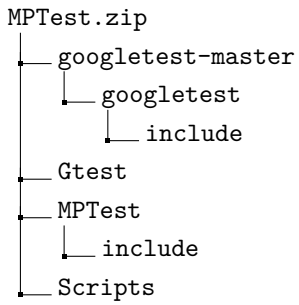
```
#include <gtest/gtest.h>
#include "Number.h"

using namespace std;

int main(int argc, char **argv) {
::testing::InitGoogleTest(&argc, argv);

return RUN_ALL_TESTS();
}
```

3. Integración con NetBeans

Esta sección hace un boceto de cómo sería la implementación de TDD en NetBeans, usando para ello GoogleTest, pero se pueden encontrar videotutoriales completos sobre este tema en Google Drive [6].

3.1. Preparandolo todo antes de empezar

3.1.1. El espacio de trabajo



Este es el material inicialmente entregado a los alumnos en Prado, clasificado por carpetas. **NetBeans** Carpeta raíz desde la que se van a crear todos los proyectos de la asignatura. **googletest-master** Rama principal de desarrollo de GoogleTest en Git. No es necesario que sea la última. **GTest** Proyecto de NetBeans que permite generar, a cada alumno, una versión enlazable de GoogleTest para su máquina y sistema operativo. **MPTest** Extensión para cubrir los tests de integración en MP. Por ahora esta extensión es un único fichero de cabeceras el cual contiene las macros necesarias para ello. **Scripts** Una serie de scripts Bash de apoyo a las funciones de NetBeans. Se recomienda empezar por crear la carpeta NetBeans y descomprimir el fichero de material previo **MPTest.zip** (Abrir en navegador →) dentro de ella, como muestra la figura a la izquierda. Abrir el proyecto **Gtest** y compilarlo (Clean & Bluild).

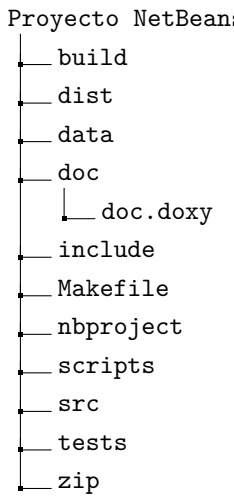


Figura 4: Estructura interna de cada proyecto NetBeans

Para tener bien ordenado el contenido de cada proyecto NetBeans, se va a dividir en las siguientes carpetas.

build Es una carpeta temporal que contiene código precompilado y pendiente de enlazar

dist Contiene los binarios ejecutables. Vamos a generar dos versiones por cada programa.

Release Es la versión más eficiente y que menos espacio ocupa. Es la que debería entregarse al cliente.

Debug Es la versión sobre la que se depuran errores y se corrigen defectos. Es más grande porque el

[6] Google Drive, Carpeta Test Driven Development (Abrir en navegador →)

binario contiene información extra para poder usar el depurador. Puede llegar a ser el doble de grande que la anterior.

data Ficheros de datos, bases de datos, ficheros de configuración que pueda necesitar el programa durante su ejecución, que no sean para testearla.

doc Aquí se almacena la documentación del proyecto, tanto del código, como la generada por los tests.

include Contiene los ficheros de cabeceras **.h**.

nbproject Es la carpeta que utiliza NetBeans para almacenar la configuración del proyecto

scritps En esta carpeta colocaremos las scripts de apoyo a Netbeans que se han desarrollado en esta asignatura y se puede ampliar con otras más.

src Contiene los principales ficheros fuente del proyecto **.cpp**, sin incluir los ficheros de test, que también son **.cpp** , que van en la siguiente carpeta

tests Carpeta dedicada a los tests unitarios y de integración. Dentro de ella se encuentran, en el primer nivel, los ficheros fuente de los tests **.cpp**, la carpeta `output` que se utiliza como carpeta temporal durante los tests y la carpeta `validation` que se utiliza para **ficheros de datos únicamente usados en la validación de la aplicación** completamente integrada.

zip Carpeta para copias de seguridad del proyecto.

3.1.2. Configurar el proyecto en NetBeans

Se recomiendo repasar primero los videotutoriales completos sobre este tema en Google Drive (Abrir en navegador →) y descargar el fichero de material **MyVectorProject.zip** (Abrir en navegador →) cuyo contenido es el siguiente:

```
MyVectorProject.zip
├── 01_Unit_MyVector.cpp (⟶ ./tests/)
├── 02 Unit_MyVector.cpp (⟶ ./tests/)
├── 10_Integration_MyVector.cpp (⟶ ./tests/)
├── documentation.doxy (⟶ ./doc/)
├── main.cpp (⟶ ./src/)
├── myvector.cpp (⟶ ./src/)
├── myvector.h (⟶ ./include/)
├── ReadmeTests.md (⟶ ./doc/)
└── scripts (⟶ ./script/)
```

1. Crear el proyecto en NetBeans como `C++ Application`.
   a) Eliminar el entorno de trabajo `Release` y dejar sólo `Debug`. Aunque en adelante se crearán otros entornos de trabajo, en lo siguiente se trabajará en `Debug`.
2. Crear la carpeta `./scripts`, copiar la script `runUpdate.sh` dentro y ejecutarla. Esto crea la estructura de carpetas descrita anteriormente y actualiza las scripts que pudiese haber nuevas.
3. Colocar cada fichero en su sitio, según indica la imagen anterior.
4. Propiedades de proyecto. Compilador
   a) Poner el estándar a C++14
   b) Añadir los `Include Directories ...` siguientes:
   del propio proyecto
   `./include`
   y las de testeo
   `../MPTest/include`
   `../googletest-master/googletest/include`

   y el resto de librerías externas que pueda utilizar el proyecto.
5. Desde la vista lógica del proyecto.
   a) En `Header Files`, `Add existing item` añadir el fichero `myvector.h`.
   b) En `Source Files`, `Add existing item` añadir los ficheros `main.cpp` y `myvector.cpp`.
6. Sobre el nombre del proyecto, botón derecho `Set As Main Project` o desde el menú principal `Run` – `Set Main Project` y seleccionar este proyecto.
7. Con esto se puede ejecutar el proyecto normalmente. Obviamente, no hace nada porque está vacío y no se llama a ninguna función. A partir de aquí empezaría el desarrollo del proyecto, el cual cada programador seguirá un orden determinado y, probablemente, diferente a todos los demás, hasta que la aplicación esté terminada. En ese momento habrá que probarla, para ver que todo funciona como se espera, en un proceso que, de nuevo, dependerá de cada programador en concreto.

3.1.3. Configurar el entorno de testeo

Sin embargo, si se sigue la metodología TDD, este proceso es bien diferente, y se lleva a cabo de manera ordenada y sistemática.

1. En la vista lógica del proyecto, `Test Files` – `New Test Folder` y le ponemos un nombre, por ejemplo `TestsMyVector`.
2. `Test Files` – `Propiedades`.
   - En `C++ Compiler` – `Preprocessor Definitions` Añadir `__INTEGRATION__`
   - En `Linker` – `Libraries ...` – `Add Project` Añadir la carpeta principal del proyecto `GTest`. En `Additional Options ...` añadir `-pthread`.
3. `TestsMyVector` – `Add Existing Item ...` Añadir los tres ficheros **.cpp** que colocamos en la carpeta de tests `01_Unit_MyVector.cpp` – `02_Unit_MyVector.cpp` – `10_Integration_MyVector.cpp`.
4. Y ya estamos en posición de comenzar las pruebas, sí, con el proyecto aún sin implementar, TDD nos irá guiando. La primera opción es ejecutar los tests desde los controles de NetBeans, por ejemplo
   a) `Run` – `Test Project`
   b) Sobre la carpeta de tests `Tests folder` – Botón derecho y `Test`
   c) Desde una terminal del proyecto, ejecutar `make test`.
5. Definir el entorno Release.
   a) Configuraciones. Duplicar la configuración `Debug` y llamarla `Release`.
   b) C++ Compiler. Preprocessor Definitions. Añadir `__RELEASE__`. Poner “Development mode” en `Release`.

4. Extensión de GoogleTest para la definición de tests de integración: MPTests

Esta sección describe las macros que se han definido en `MPTests.h` para cubrir algunas deficiencias de GoogleTest respecto a una asignatura como Metodología de la Programación.

4.1. Saltar un test

GoogleTest pasa siempre todos los tests que hayamos definido. Así, si tenemos un cojunto de tests muy amplio y falla el primero, esto no hace que GoogleTest se interrumpa, sino que sigue ejecutando todos los tests, con la consiguiente secuencia de fallos encadenados.

Por ejemplo, si nuestro constructor de la clase `Number` no estuvise bien definido y tuviésemos otro test adicional, el fallo del primero se acumularía al fallo del segundo (ver Figura 5).

```
TEST(MAIN, MiniClass_get) {
Number n;
ASSERT_EQ(n.get(), 0) << "Una instancia recién creada de Number, siempre devuelve el
    número 0" << endl;

}

TEST(Number, Number_increase) {
Number a;       // Se inicializa a 0
a.increase();   // Se incrementa en 1
ASSERT_EQ(a.get(),1); <<"Si incrementamos una instancia recién creada de Number, el
    resultado debe ser 1" << endl;
}
```

Esto, en sí, no es un problema, pero sí es molesto, porque difumina el test que falla y se puede perder un poco el foco. Para ello, se han definido aserciones personalizadas en `MPTest` que, en cuanto falla un test, se salta (`SKIP_`) todos los demás y no genera más mensajes de error, permitiendo enfocar mejor el único tests que ha fallado.

```
TEST(MAIN, MiniClass_get) {
Number n;
SKIP_ASSERT_EQ_R(n.get(), 0) << "Una instancia recién creada de Number, siempre
    devuelve el número 0" << endl;

}
```

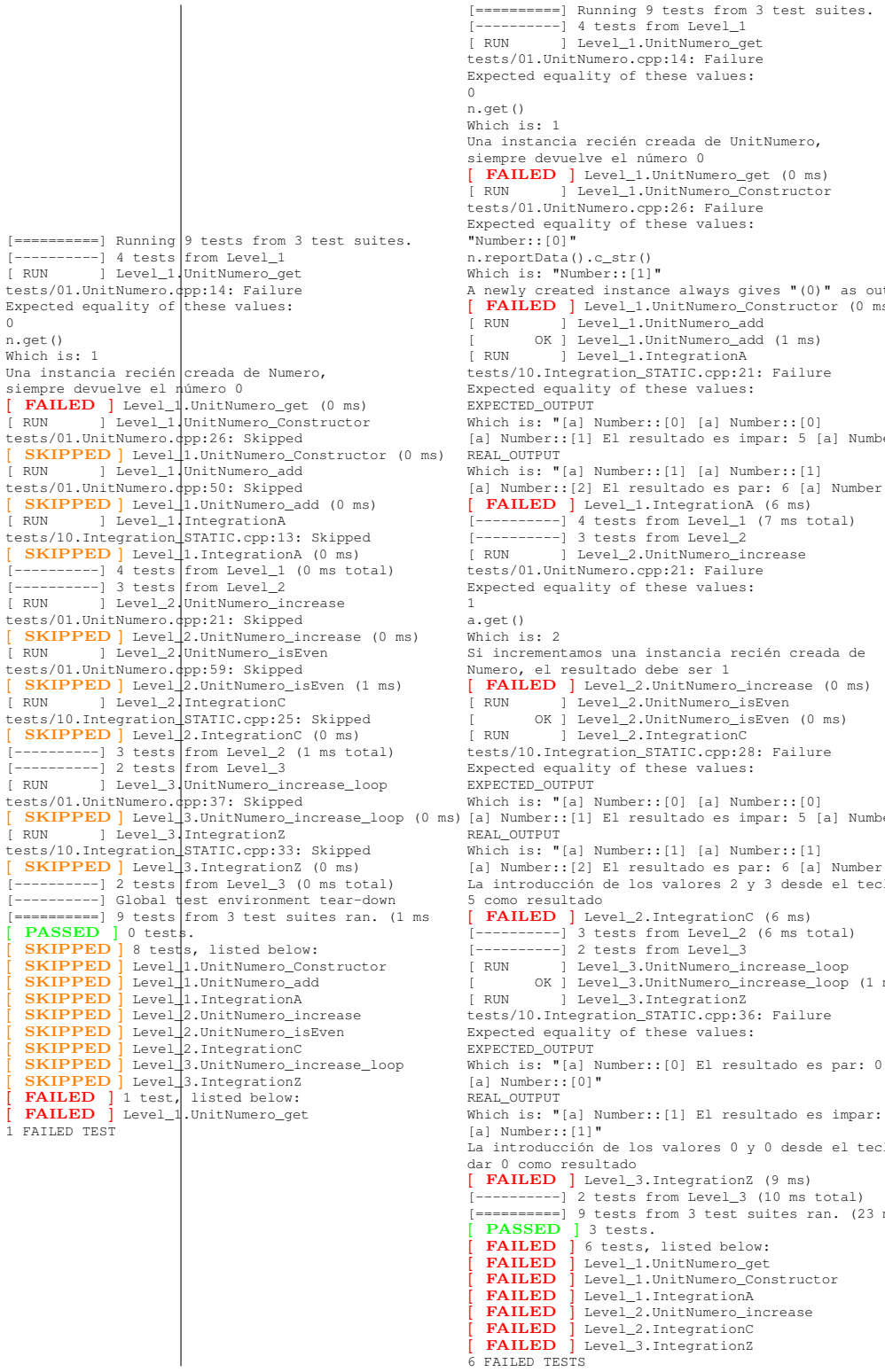


Figura 5: Diferencias entre el uso de las aserciones `SKIP_ASSERT.*` (a la izquierda) y las habituales de GoogleTest `ASSERT_*` (a la derecha).

4.2. Tests de integración

Los tests unitarios se centran en comprobar que las unidades funcionales hacen lo que se espera que hagan, pero no comprueban el funcionamiento de `main()`. Esto es un problema muy abierto para el que existen múltiples plataformas, la mayoría de ellas demasiado complejas como para ser introducidas en el marco de una asignatura de primero de grado como es Metodología de la Programación. No obstante, dada la base de confianza que proporcionan los tests unitarios y, a partir de las experiencias previas en esta asignatura en DGIIM, se pueden realizar algunos tests de integración relativamente sencillos, que permiten ejecutar los programas completos, con su verdadero main, contra una serie de datos de entrada controlados por el profesor, y de los cuales se conoce la salida esperada. Como el siguiente programa.

```
int main() {
Number a;
int increm1, increm2;
cout << "Introduzca dos incrementos consecutivos ";
cin >> increm1>>increm2;
for (int i=0;i<increm1;a.increase(), i++);
for (int i=0;i<increm2;a.increase(),i++);
if (a.isEven()) {
cout << "El resultado es par: "<< a.get()<<endl;
} else {
cout << "El resultado es impar: "<< a.get() << endl;
}
}

$$ Introduzca dos incrementos consecutivos 2 3
$$ El resultado es impar: 5
```

Para ello, se ejecuta el programa y se captura parte de su salida, marcando en esta salida, los checkpoints que se consideren necesarios y permitiendo la libertad del alumno para programar sus propias salidas, en los que se analiza, tanto la salida del programa, como el estado interno de cada dato del programa que pueda ser relevante. Finalmente, el test de integración comprueba que la salida esperada coincide con la salida detectada real en esos checkpoints.

4.2.1. Definir el entorno de ejecución controlado para testear la aplicación completa

La macro `DEF_EXECUTION_ENVIRONMENT(<LABEL>)` permite crear un entorno controlado para la ejecución de la aplicación, en la que se puede simular la redirección de entrada y la de salida para comprobar distintos casos de ejecución. Cada caso se programaría en un test diferente. la etiqueta debería coincidir con el nombre del test (segundo parámetro en la llamada a `TEST(C,A)`).

La macro `FROM_KEYBOARD()` permite definir una entrada simulada desde el teclado, la cual, al llamar al binario con la macro `CALL_FROM_KEYBOARD()` simula esa entrada desde `cin` al ejecutar el binario.

```
TEST(MAIN, IntegrationA) {
DEF_EXECUTION_ENVIRONMENT(IntegrationA);
FROM_KEYBOARD("2 3");
CALL_FROM_KEYBOARD( "");
}
```

En caso de que la simulación de los datos de entrada desde teclado sean muy largas como para ponerlas en una línea con `FOM_KEYBOARD()` se podría grabar esta secuencia de datos en un fichero con extensión input en la carpeta `./tests/validation/`. Nótese que el nombre del fichero debe coincidir con el nombre del test, en este caso "IntegrationA";

```
$$  echo "2 3" > ./tests/validation/IntegrationA.input

TEST(MAIN, IntegrationA) {
DEF_EXECUTION_ENVIRONMENT(IntegrationA);
CALL_FROM_FILE(IntegrationA, "");
}
```

Esto hace que se ejecute el programa completamente, pero su salida, ni aparece en la pantalla durante el test, ni podemos comprobar que es la esperada.

4.2.2. Definiendo checkpoints en la salida

El alumno puede programar su salida con `cout` como individualmente prefiera, pero puede incluir checkpoints para validar los tests de integración mediante la macro `CVAL`. Cuando se está en testeo, esta macro se expande a `cerr`, en otro caso, se expande a `cout`, lo que permite capturar la salida del programa, sólo en aquellos puntos que se desee.

```
int main() {
Number a;

int increm1, increm2;
cout << "Introduzca dos incrementos consecutivos ";
cin >> increm1>>increm2;
for (int i=0;i<increm1;a.increase(), i++);
for (int i=0;i<increm2;a.increase(),i++);
if (a.isEven()) {
CVAL << "El resultado es par: "<< a.get()<<endl;
} else {
CVAL << "El resultado es impar: "<< a.get() << endl;
}
}
```

```
TEST(MAIN, IntegrationA) {
DEF_EXECUTION_ENVIRONMENT(IntegrationA);
FROM_KEYBOARD("2 3");
EXPECTED_TEXT_OUTPUT("El resultado es impar: 5");
CALL_FROM_KEYBOARD( "")<<"La introducción de los valores 2 y 3 desde el teclado debe
    dar 5 como resultado"<<endl;
}

TEST(MAIN, IntegrationB) {
DEF_EXECUTION_ENVIRONMENT(IntegrationB);
FROM_KEYBOARD("2 2");
EXPECTED_TEXT_OUTPUT("El resultado es par: 4");
CALL_FROM_KEYBOARD("")<<"La introducción de los valores 2 y 2 desde el teclado debe
    dar 4 como resultado"<<endl;;
}
```

Para ello, la macro `EXPECTED_TEXT_OUTPUT` define cómo debería de haber sido la salida en el caso de que fuese correcta, eliminando espacios duplicados y saltos de línea. Esta salida esperada se contrasta contra la salida real en la última llamada `ASSERT_STREQ` que comprueba si ambas salidas, especificadas como secuencias de caracteres, coinciden o no.

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from MAIN
[ RUN      ] MAIN.IntegrationA
[       OK ] MAIN.IntegrationA (7 ms)
[ RUN      ] MAIN.IntegrationB
[       OK ] MAIN.IntegrationB (6 ms)
[----------] 2 tests from MAIN (13 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (13 ms total)
[  PASSED  ] 2 tests.

RUN FINISHED; exit value 0; real time: 20ms; user: 0ms; system: 10ms
```

Hay que tener mucho cuidado con estos tests de integración pues hacen una comparación carácter a carácter de forma estricta por lo que conviene saber exactamente cómo se produce la salida a través de `CVAL`. En cualquier caso, cuando la salida esperada difiere de la salida real obtenida, `MPTest` muestra la salida tal y como debía haber sido, y marca las diferencias con la sintaxis del comando `wdiff`[7]. Por ejemplo, si la salida esperada debería haber sido

```
"A B C D E F G"
```

entonces las siguientes salidas producirían las siguientes diferencias

Salida real	**wdiff**
`""`	`"[-A B C D E F G-]"`
`"A B C D "`	`"A B C D [-E F G-]"`
`"A B C D F G"`	`"A B C D [-E-] F G"`
`"A B C D H F G"`	`"A B C D [-E-] {+H+} F G"`
`"1 2 3 4 5 6 7"`	`"[-A B C D E F G-]{+1 2 3 4 5 6 7+}"`

En esta sintaxis, las palabras que deberían estar en la salida, pero no lo están, se marcan con `[- <palabra> -]` y las que sí están en la salida, pero sobran, se marcan con `{+ <palabra> +}`

Por ejemplo, si el programa anterior estuviese mal y tuviese los casos par e impar intercambiados por error

```
int main() {
Number a;

int increm1, increm2;
cout << "Introduzca dos incrementos consecutivos ";
cin >> increm1>>increm2;
for (int i=0;i<increm1;a.increase(), i++);
for (int i=0;i<increm2;a.increase(),i++);
if (!a.isEven()) { // Error: está al revés
CVAL << "El resultado es par: "<< a.get()<<endl;
} else {
CVAL << "El resultado es impar: "<< a.get() << endl;
}
}
```

Entonces la salida del test hubiese sido también diferente, se puede ver que el valor calculado es correcto, pero las palabras par e impar aparecen intercambiadas.

[7] Para ello debe de estar instalado el programa `wdiff`

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from MAIN
[ RUN      ] MAIN.IntegrationA
tests/10.Integration_STATIC_Latex.cpp:34: Failure
Expected equality of these values:
""
_wdiff.c_str()
Which is: "El resultado es [-impar:-] {+par:+} 5"
La introducción de los valores 2 y 3 desde el teclado debe dar 5 como resultado

[  FAILED  ] MAIN.IntegrationA (13 ms)
[ RUN      ] MAIN.IntegrationB
tests/10.Integration_STATIC_Latex.cpp:41: Failure
Expected equality of these values:
""
_wdiff.c_str()
Which is: "El resultado es [-par:-] {+impar:+} 4"
La introducción de los valores 2 y 2 desde el teclado debe dar 4 como resultado

[  FAILED  ] MAIN.IntegrationB (14 ms)
[----------] 2 tests from MAIN (27 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (27 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 2 tests, listed below:
[  FAILED  ] MAIN.IntegrationA
[  FAILED  ] MAIN.IntegrationB
```

4.2.3. Aplicaciones embucladas

Es muy frecuente que, debido a errores de programación, se creen ciclos infinitos en la ejecución, lo que llevaría a un bloqueo de todos los tests. Para resolver este problema, todas las aplicaciones se ejecutan con un time-out, por defecto en 30 segundos, transcurridos los cuales, se interrumpe la aplicación y el test se da por fallado. En caso de aplicaciones que necesiten más tiempo para una ejecución normal, se puede personalizar este time-out con la macro `SET_TIMEOUT_S` que especifica un tiempo de ejecución máximo en segundos.

```
TEST(UnitNumber,IntegrationFrozen) {
DEF_EXECUTION_ENVIRONMENT(IntegrationFrozen);
FROM_KEYBOARD("2 2147483647");
SET_TIMEOUT_S(3);
EXPECTED_TEXT_OUTPUT("El resultado es par: 4");
CALL_FROM_KEYBOARD("")<<"La introducción de los valores 2 y 2 desde el teclado debe
    dar 4 como resultado"<<endl;;
}
}
```

4.2.4. Validando el contenido de ciertas clases

Además, se pueden aprovechar los checkpoints de salida controlada para serializar ciertas clases y capturarlas durante la ejecución, en los puntos que se estime necesario. Para ello se usa la combinación de la macro `REPORT_DATA(x)` (Figura 6) que vuelca en `CVAL` la salida del método `report_data()`, el cual debe de estar instanciado en la clase a la que pertenece el objeto `x`.

Cuando no está en testeo, esta última macro desaparece y no hace nada para no interferir con la salida real de la aplicación (Figura 7). En ciertas clases cuya serialización sea muy extensa, el método `reportData()` puede generar un hash, un código encriptado, que representa dicha serialización, pudiendo compararse la serialización esperada con la real sin más que comparar estos códigos.

```
class Number {
private:
int data;
public:

Number() { data = 0; }
int get() const { return data; }
int increase() {data++;}
void add(Number n) {data += n.get();}
bool isEven() {return data %2 == 0;}
/// Serialización de la clase para inspeccionar su contenido en determinados puntos
 del programa
std::string reportData() {return "{Number}::["+std::to_string(data)+"]";}
};
```

(a)

```
int main() {
Number a;
REPORT_DATA(a);      /// Chekpoint inicial
int increm1, increm2;
cout << "Introduzca␣dos␣incrementos␣consecutivos␣";
cin >> increm1>>increm2;
for (int i=0;i<increm1;a.increase(), i++) REPORT_DATA(a);    /// Checkpoint continuo
for (int i=0;i<increm2;a.increase(),i++);
if (a.isEven()) {
CVAL << "El␣resultado␣es␣par:␣"<< a.get()<<endl;
} else {
CVAL << "El␣resultado␣es␣impar:␣"<< a.get() << endl;
}
REPORT_DATA(a);      /// Checkpoint final
}
```

(b)

```
[a] {Number}::[0]
Introduzca dos incrementos consecutivos 4 1
[a] {Number}::[0]
[a] {Number}::[1]
[a] {Number}::[2]
[a] {Number}::[3]
El resultado es impar: 5
[a] {Number}::[5]
}
```

(c)

```
TEST(UnitNumber,IntegrationA) {
DEF_EXECUTION_ENVIRONMENT(IntegrationA);
EXPECTED_OUTPUT(
[a] {Number}::[0]
[a] {Number}::[0]
[a] {Number}::[1]
El resultado es impar: 5
[a] {Number}::[5]);
CALL_FROM_FILE("");
}
```

(d)

Figura 6: (a) Fichero `Number.h` con el método `reportData()` para examinar su contenido detallado durante la ejecución, en caso de que fuese necesario. (b) Función `main()` en la que se han puesto varios puntos de control para examinar en detalle la clase `Number` conforme avanza el programa. (c) Salida del programa que muestra la evolución del interior de la clase `Number` a lo largo de la ejecución del programa.(d) Test de integración que comprueba que, tanto la salida como la evolución de la clase son correctas.

```
Introduzca dos incrementos consecutivos 4 1
El resultado es impar: 5
```

Figura 7: Salida del mismo programa de la Figura 6 en el modo `Release`, en el que se puede apreciar que las salidas de validación han desaparecido

```
TEST(UnitNumber,IntegrationC) {
DEF_EXECUTION_ENVIRONMENT(IntegrationC);
FROM_KEYBOARD("2 3");
USE_MEMORY_LEAKS(__VALGRIND__);
EXPECTED_TEXT_OUTPUT( "[a] {Number}::[0] [a] {Number}::[1] [a] {Number}::[2] El
 resultado es impar: 5 [a] {Number}::[5] ");
CALL_FROM_KEYBOARD("")<<"La introducción de los valores 2 y 3 desde el teclado debe
 dar 5 como resultado"<<endl;;
}
```

Figura 8: Incorporación de chequeo de memoria dinámica en un test, pudiendo utilizar, si están instalados, los paquetes `__VALGRIND__` o `__DRMEMORY__`

4.2.5. Detectando problemas de memoria dinámica

Otra de las propiedades de la extensión `MPTest` es que permite pasar un control de memmoria dinámica, integrado y homogéneo, mediante el uso del entorno **Valgrind**[8] o **DrMemory**[9]

Para ello se dispone de la macro `USE_MEMORY_LEAKS(__VALGRIND__)` o `USE_MEMORY_LEAKS(__DRMEMORY__)` que están integradas en el entorno de ejecución y proporcionan la misma salida en GoogleTest, para ambas herramientas, de los errores más comunes. En este caso, la aplicación es suficientemente sencilla como para no producir errores en un test básico como el mostrado en la Figura 9.

La inclusión del chequeo de memoria dinámica es muy sencilla. Tan solo hay que solicitarla y esperar al resultado. En primer lugar se pasa el chequeo de memoria, porque es lo más prioritario. Si este falla, todo el test falla, en otro caso, se pasa el test en sí mismo con la validación de datos. El resultado aparece integrado en la salida de GoogleTest como un test más que se identifica como "MEMCHECK" e indica el paquete con el que se ha realizado.

[8]Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

The Valgrind distribution currently includes seven production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and two different heap profilers. It also includes an experimental SimPoint basic block vector generator. It runs on the following platforms: X86/Linux, AMD64/Linux. https://www.valgrind.org/

[9]Dr. Memory is a memory monitoring tool capable of identifying memory-related programming errors such as accesses of uninitialized memory, accesses to unaddressable memory (including outside of allocated heap units and heap underflow and overflow), accesses to freed memory, double frees, memory leaks, and (on Windows) handle leaks, GDI API usage errors, and accesses to un-reserved thread local storage slots. Dr. Memory operates on unmodified application binaries running on Windows, Linux, Mac, or Android on commodity IA-32, AMD64, and ARM hardware. (Abrir en navegador →)

```
[==========] Running 9 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 9 tests from UnitNumber
[ RUN      ] UnitNumber.UnitNumber_get
[       OK ] UnitNumber.UnitNumber_get (0 ms)
[ RUN      ] UnitNumber.UnitNumber_increase
[       OK ] UnitNumber.UnitNumber_increase (1 ms)
[ RUN      ] UnitNumber.UnitNumber_Constructor
[       OK ] UnitNumber.UnitNumber_Constructor (0 ms)
[ RUN      ] UnitNumber.IntegrationC
[ MEMCHECK ] IntegrationC-valgrind
[       OK ] IntegrationC-valgrind
[       OK ] UnitNumber.IntegrationC (1024 ms)
```

Veamos qué ocurre si se poducen errores de memoria poniendo un main más complejo. El siguiente `main()` produce casi todos los tipos de error que se van a considerar:

NO_INIT Uso de memoria no inicializada.

BAD_READ Acceso de lectura a una posición de memoria no permitida.

BAD_WRITE Acceso de escritura a una posición de memoria no permitida.

MEM_LEAK Pérdida de memoria debido a que no se ha liberado antes de terminar el programa.

BAD_FREE Uso del operador inadecuado para liberar la memoria.

```
int main() {
    Number *a;
    a = new Number [SIZE];

    REPORT_DATA((*a)); /// @brief
Chekpoint inicial
    int increm1, increm2, increm3;
    cout << "Introduzca dos incrementos
consecutivos ";
    cin >> increm1>>increm2;
    for (int i = 0; i < increm1; a[1].
increase(), i++) REPORT_DATA((*a)); ///
 @brief Checkpoint continuo
    for (int i = 0; i < increm3; a[0].
increase(), i++);
    if (a[0].isEven()) {
    CVAL << "El resultado es par: " << a
[0].get() << endl;
    } else {
    CVAL << "El resultado es impar: " <<
 a[0].get() << endl;
    }
    REPORT_DATA((*a)); /// @brief
Checkpoint final
    //delete [] a;
    }
```

```
[==========] Running 9 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 6 tests from UnitNumber
[ RUN      ] UnitNumber.UnitNumber_get
[       OK ] UnitNumber.UnitNumber_get (0 ms)
[ RUN      ] UnitNumber.UnitNumber_increase
[       OK ] UnitNumber.UnitNumber_increase (0 ms)
[ RUN      ] UnitNumber.UnitNumber_Constructor
[       OK ] UnitNumber.UnitNumber_Constructor (0 ms)
[ RUN      ] UnitNumber.UnitNumber_increase_loop
[       OK ] UnitNumber.UnitNumber_increase_loop (0 ms)
[ RUN      ] UnitNumber.UnitNumber_add
[       OK ] UnitNumber.UnitNumber_add (0 ms)
[ RUN      ] UnitNumber.UnitNumber_isEven
[       OK ] UnitNumber.UnitNumber_isEven (0 ms)
[----------] 6 tests from UnitNumber (2 ms total)

[----------] 3 tests from IntegrationNumber
[ RUN      ] IntegrationNumber.IntegrationA
[ MEMCHECK ] IntegrationA-valgrind
[--BAD_READ] (main_Heap.cpp:21) (main_Heap.cpp:49) (main_Heap.cpp:43)
[-BAD_WRITE] (main_Heap.cpp:21) (main_Heap.cpp:49) (main_Heap.cpp:43)
[---NO_INIT] (main_Heap.cpp:50)
[--MEM_LEAK] (main_Heap.cpp:43)
src/main_Heap.cpp:126: Failure
Failed
[  FAILED  ] IntegrationNumber.IntegrationA (1085 ms)
[ RUN      ] IntegrationNumber.IntegrationC
src/main_Heap.cpp:130: Skipped

[  SKIPPED ] IntegrationNumber.IntegrationC (0 ms)
[ RUN      ] IntegrationNumber.IntegrationZ
src/main_Heap.cpp:139: Skipped

[  SKIPPED ] IntegrationNumber.IntegrationZ (0 ms)
[----------] 3 tests from IntegrationNumber (1085 ms total)

[----------] Global test environment tear-down
[==========] 9 tests from 2 test suites ran. (1087 ms total)
[  PASSED  ] 6 tests.
[  SKIPPED ] 2 tests, listed below:
[  SKIPPED ] IntegrationNumber.IntegrationC
[  SKIPPED ] IntegrationNumber.IntegrationZ
[  FAILED  ] 1 test, listed below:
[  FAILED  ] IntegrationNumber.IntegrationA

1 FAILED TEST
```

Figura 9: Izquierda: Código con varios errores de gestión de memoria. Derecha: Informe de errores incrustado en GoogleTest mostrando el error y las líneas de código afectadas.

En este caso, se puede ver en la Figura 9 el informe de GoogleTest al que estamos habituados, arriba a la derecha, el cual sigue el mismo formato con ambas herramientas de análisis de memoria.

4.2.6. Paso de parámetros a main

Para terminar, no sólo se pueden redireccionar la entrada y la salida desde el mismo entorno de Test para comprobar que la aplicación funciona como se esperaba, sino que se le pueden pasar argumentos a main desde la misma llamada del test, para completar todas las posibilidades de testeo de integración que cubre `MPTest` y que son las que se desarrollan en las asignaturas Fundamentos de la Programación y Metdología de la Programación.

```
TEST(INTEGRATION, CALL_ERROR_BAD_OPTION) {

DEF_EXECUTION_ENVIRONMENT(CALL_ERROR_BAD_OPTION);
EXPECTED_TEXT_OUTPUT("Error in call");
CALL("-l ES -r 2020 -k")<< "An unexpected parameter in the command line must produce
    \"Error in call\""<< endl;
}
```

4.3. Reportes de tests

Además de todo lo anterior, la extensión `MPTest` genera varios informes.

4.3.1. Reporte inicial de los tests que hay preparados

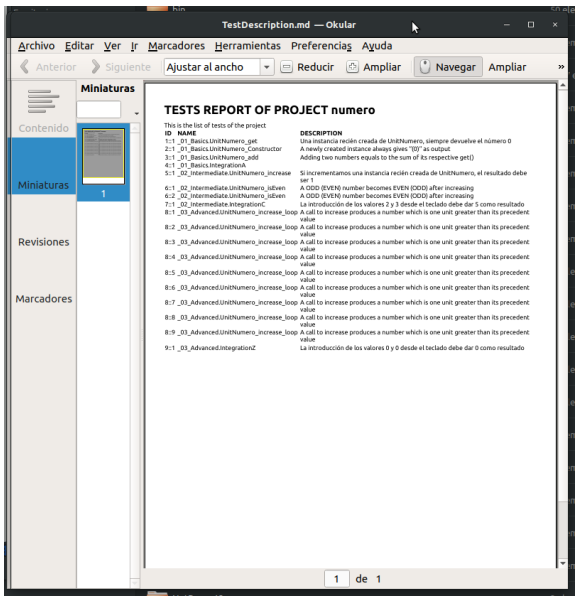


Figura 10: Informe preliminar de tests que se van a pasar.

La extensión `MPtest` puede generar un reporte inicial que sirve de guía durante el proceso de testeo. Este informe contiene todos los tests que haya activos en NetBeans, con la misma numeración que saldrán en la pantalla y que el reporte anterior, a modo de guía para el testeo, pero antes de pasar los tests. Este test hay que generarlo sólo una vez, y se hace manualmente desde una consola del proyecto en la que se invoca a los tests.

```
make clean
make CXXFLAGS=-D__REPORT_ALL_TESTS__ test
make clean
```

Este reporte se genera en

`./doc/markdown/TestDescription.md`

se puede visualizar con un visor de PDFs como `okular` (ver Figura 10).

4.3.2. Reporte del resultado de pasar los tests

El primero es un informe en formato Markdown que describe el resultado auditado de los tests, por si hubiese que justificarlo. Este informe se encuentra en la carpeta

`./tests/output/TestReport.md`

y se puede visualizar con un visor de PDFs como `okular` (ver Figura 11).

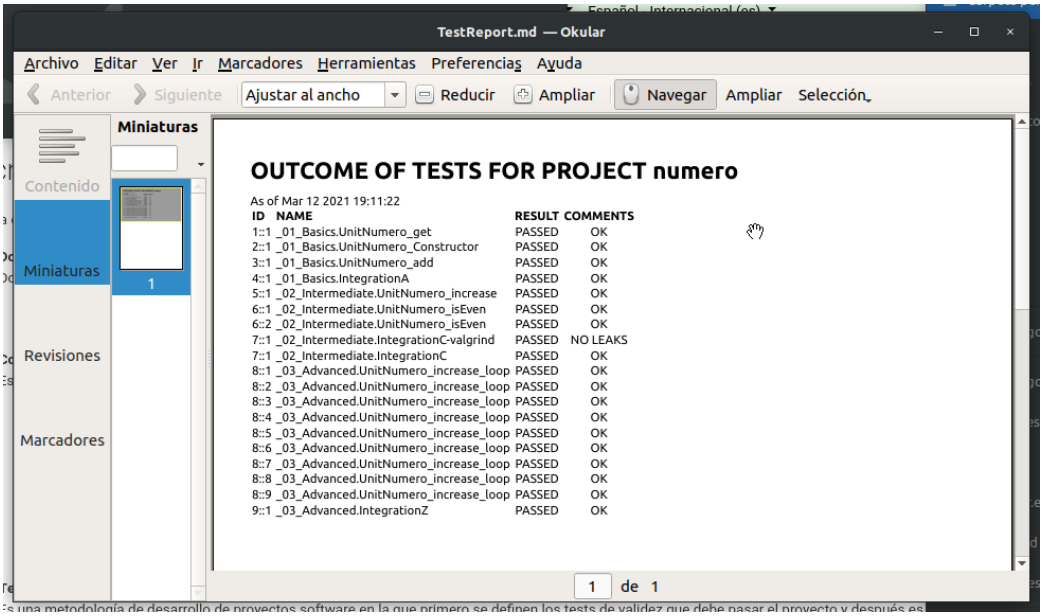


Figura 11: Reporte de tests pasados. Una vez terminado el proceso de testeo, se genera un informe de auditoría de tests situado en `./tests/output/TestReport.md`. La primera columna muestra dos números, el primero es el ordinal del test, coincidiendo con el ordinal de Googletest (ver Figura 5). Cada test puede tener varios chequeos en su interior, para ello, el segundo índice hace referencia al orden en el que aparece este chequeo.

Este reporte muestra todos los tests, tanto unitarios como de integración, que se han preparado y su resultado. Obsérvese que el test de la Figura 8, cuyo nombre es `IntegrationC` aparece varias veces, una de ellas, la primera, es el chequeo de memoria (PASSED - NO LEAKS), y la otra es el test en sí (PASSED - OK).

La Figura 12 muestra sendos reportes para los problemas de memoria mostrados en la Figura 9. A la izquierda el informe detallado que genera `MPTest` cuando se selecciona Valgrind y a la derecha el informe de Dr. Memory. Se puede ver que, aunque el informe en pantalla integrado con GoogleTest se ha unificado por igual con ambas herramientas (Figura 9), el informe particular puede variar, tanto en los mensajes propios de cada uno como en la detección.

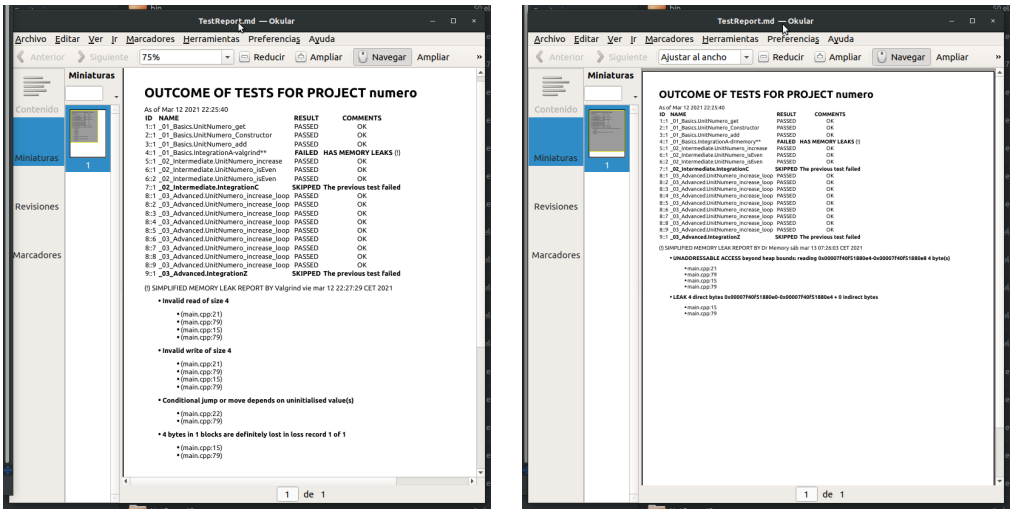


Figura 12: Informes de error generados por `MPTest` para el caso de la Figura 9, a la izquierda el de Valgrind y a a la derecha el de Dr. Memory.

5. Propiedades de TDD

Algunas propiedades de TDD que conviene subrayar son las siguientes:

1. Corrección. Los tests deben de ser correctos. Es código, así que el mismo test podría contener errores, lo que llevaría a una peligrosa validación de las unidades funcionales.

2. Completitud. El conjunto de tests unitarios debe ser completo o lo más completo posible, para cubrir todas las posibles casuísticas.

3. Legibilidad. Tanto el test como su resultado, sobre todo en caso de fallo, debe ser lo más claro posible.

4. Repetibilidad. En caso de repetir varias veces un mismo test con los mismos datos, debe producir el mismo resultado.

5. Agilidad. Cada test debe de ejecutarse en el mínimo tiempo posible y se debe ejecutar al alcance del mínimo número de clics.