Módulo 2

Entorno de Trabajo

David Vallejo Fernández
david.vallejo@tegnix.com

Carlos González Morcillo
carlos.gonzalez@tegnix.com

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

Capítulo 2

Entorno de Trabajo

Actualmente, existen un gran número de aplicaciones y herramientas que permiten a los desarrolladores de videojuegos, y de aplicaciones en general, aumentar su productividad a la hora de construir software, gestionar los proyectos y recursos, así como automatizar procesos de construcción. En este capítulo, se pone de manifiesto la importancia de la gestión en un proyecto software y se muestran algunas de las herramientas de desarrollo más conocidas en este ámbito.

Así mismo, también se discute cómo instalar y configurar el framework NME en el sistema operativo Ubuntu y cómo generar un ejecutable de un primer proyecto sencillo para GNU/Linux y Android.

2.1. Procesos básicos de desarrollo

En la construcción de software no trivial, las herramientas de gestión de proyectos y de desarrollo facilitan la labor de las personas que lo construyen. Conforme el software se va haciendo más complejo y se espera más funcionalidad de él, se hace necesario el uso de herramientas que permitan **automatizar** los procesos del desarrollo, así como la gestión del proyecto y su documentación.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

Figura 2.1: El proyecto GNU proporciona una gran abanico de herramientas de desarrollo y son utilizados en proyectos software de todo tipo.

Además, dependiendo del contexto, es posible que existan otros integrantes del proyecto que no tengan formación técnica y que necesiten realizar labores sobre el producto como traducciones, pruebas, diseño gráfico, etc.

Los videojuegos son proyectos software que, normalmente, requieren la participación de varias personas con diferentes perfiles profesionales (programadores, diseñadores gráficos, compositores de sonidos, etc.). Cada uno de ellos, trabaja con diferentes tipos de datos en diferentes tipos de formatos. Todas las herramientas que permitan la construcción automática del proyecto, la **integración** de sus diferentes componentes y la coordinación de sus miembros serán de gran ayuda en un entorno tan heterogéneo.

Desde el punto de vista de la gestión del proyecto, una tarea esencial es la automatización del proceso de compilación y de construcción de los programas. Una de las tareas que más frecuentemente se realizan mientras se desarrolla y depura un programa es la de compilación y construcción. Cuanto más grande y complejo sea un programa, mayor es el tiempo que se pierde en esta fase. Por tanto, un proceso automático de construcción de software ahorrará muchas pérdidas de tiempo en el futuro.

En los sistemas GNU/Linux es habitual el uso de herramientas como el compilador GCC, el sistema de construcción GNU Make y el depurador GDB. Todas ellas creadas en el proyecto GNU y orientadas a la creación de programas en C/C++, aunque también pueden ser utilizadas con otras tecnologías. También existen editores de texto como GNU Emacs o vi, y modernos (pero no por ello mejores) entornos de desarrollo como Eclipse que no sólo facilitan las labores de escritura de código, sino que proporcionan numerosas herramientas auxiliares dependiendo del tipo de proyecto. Por ejemplo, Eclipse puede generar los archivos Makefiles necesarios para automatizar el proceso de construcción con GNU Make.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Figura 2.2: GCC es una colección de compiladores para lenguajes como C/C++ y Java.

La compilación y la depuración y, en general, el **proceso de construcción** es una de las tareas más importantes desde el punto de vista del desarrollador de aplicaciones. En muchas ocasiones, parte de los problemas en el desarrollo de un programa vienen originados directa o indirectamente por el propio proceso de construcción del mismo. Hacer un uso indebido de las opciones del compilador, no depurar utilizando los programas adecuados o realizar un incorrecto proceso de construcción del proyecto son ejemplos típicos que, en muchas ocasiones, consumen demasiado tiempo en el desarrollo. Por todo ello, tener un conocimiento sólido e invertir tiempo en estas cuestiones ahorra más de un quebradero de cabeza a lo largo del ciclo de vida de la aplicación.

En esta sección se estudia la terminología y los conceptos básicos en el ámbito de los procesos de construcción de aplicaciones. Como caso particular de estudio inicial, se muestra el uso del compilador de C/C++ GCC, el depurador GDB y el sistema de construcción automático GNU Make.

Esta primera toma de contacto servirá para obtener una perspectiva general que, posteriormente, será completada con el uso de las herramientas de compilación y ejecución del **framework NME** en las secciones 2.3.3 y 2.4.

2.1.1. Código fuente, código objeto y código ejecutable

La programación consiste en escribir programas. Los **programas** son procedimientos que, al ejecutarse de forma secuencial, obtienen unos resultados. En muchos sentidos, un programa es como una receta de cocina: una especificación secuencial de las acciones que hay que realizar para conseguir un objetivo. Cómo de abstractas sean estas especificaciones define el **nivel de abstracción** de un lenguaje.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Los programas se pueden escribir directamente en **código ejecutable**, también llamado código binario o código máquina. Sin embargo, el nivel de abstracción tan bajo que ofrecen estos lenguajes haría imposible que muchos proyectos actuales pudieran llevarse a cabo. Este código es el que entiende la máquina donde se va a ejecutar el programa y es específico de la plataforma. Por ejemplo, máquinas basadas en la arquitectura PC no ofrecen el mismo **repertorio de instrucciones** que otras basadas en la arquitectura PPC o ARM. A la dificultad de escribir código de bajo nivel se le suma la característica de no ser portable.

Por este motivo se han creado los **compiladores**. Estos programas traducen **código fuente**, programado en un lenguaje de alto nivel, en el código ejecutable para una plataforma determinada. Un paso intermedio en este proceso de compilación es la generación de **código objeto**, que no es sino código en lenguaje máquina al que le falta realizar el *proceso de enlazado*.

Aunque en los sistemas como GNU/Linux la extensión en el nombre de los archivos es puramente informativa, los archivos fuente en C++ suelen tener las extensiones `.cpp`, `.cc`, y `.h`, `.hh` o `.hpp` para las cabeceras. Por su parte, los archivos de código objeto tienen extensión `.o` y lo ejecutables no suelen tener extensión.

2.1.2. Compilación

El **compilador** es el responsable del proceso de compilación, el cual es un programa que, a partir del código fuente, genera el código ejecutable para la máquina destino. Este proceso de *traducción* automatizado permite al programador, entre otras muchas ventajas:

- No escribir código de muy bajo nivel.
- Abstraerse de la características propias de la máquina tales como registros especiales, modos de acceso a memoria, etc.
- Escribir código portable. Basta con que exista un compilador en una plataforma que soporte C++ para que un programa pueda ser portado.

Aunque la función principal del compilador es la de actuar como **traductor** entre dos tipos de lenguaje, este término se reserva a los programas que transforman de un lenguaje de alto nivel a otro; por ejemplo, el programa que transforma código C++ en Java.

La figura 2.3 muestra la estructura funcional de un compilador, que representa las **fases de compilación** en las que, normalmente, está dividido dicho proceso.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

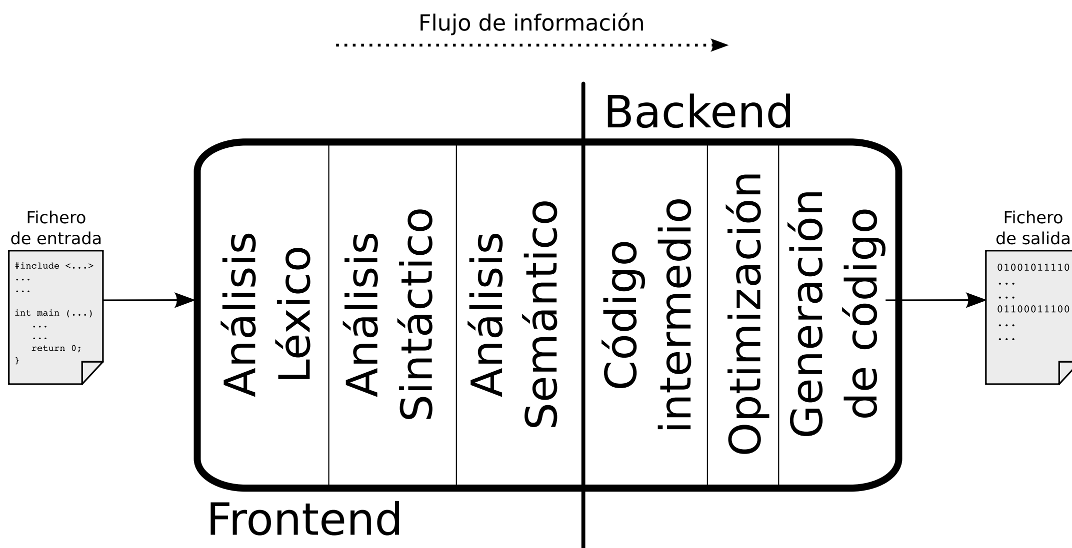

Figura 2.3: Fases del proceso de compilación.

Las fases están divididas en dos grandes bloques:

- **Frontend**: o frontal del compilador. Es el encargado de realizar el análisis léxico, sintáctico y semántico de los ficheros de entrada. El resultado de esta fase es un **código intermedio** que es independiente de la plataforma destino.
- **Backend**: el código intermedio pasa por el **optimizador** y es mejorado utilizando diferentes estrategias como la eliminación de código muerto o de situaciones redundantes. Finalmente, el código intermedio optimizado lo toma un **generador** de código máquina específico de la plataforma destino. Además de la generación, en esta fase también se realizan algunas optimizaciones propias de la plataforma destino.

En definitiva, el proceso de compilación de un compilador está dividido en etapas bien diferenciadas, que proporcionan diferente funcionalidad para las etapas siguientes hasta la generación final.

2.1.3. Caso de estudio: GNU GCC

GCC es un compilador cuya estructura es muy similar a la presentada anteriormente. Sin embargo, cada una de las fases de compilación la realiza un componente bien definido e independiente. Concretamente, al principio de la fase de compilación, se realiza un procesamiento inicial del código fuente utilizando el *preprocesador* GNU CPP, posteriormente

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

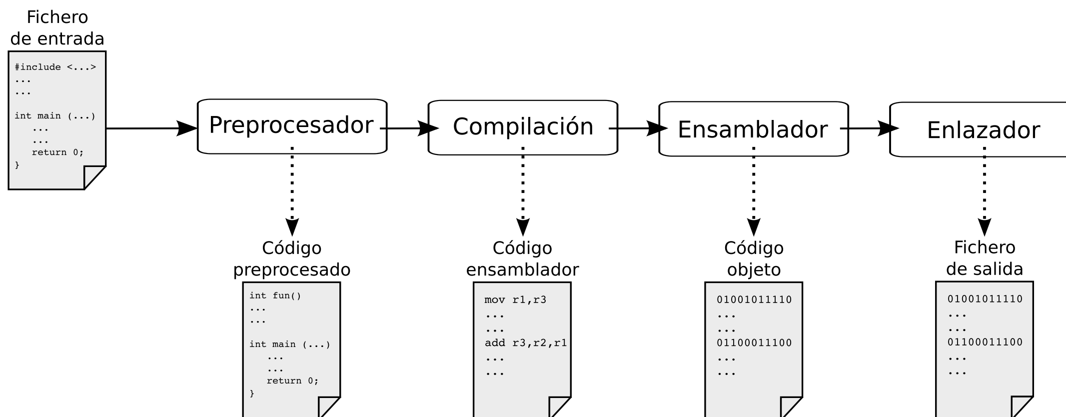

Figura 2.4: Proceso de compilación en GCC.

se utiliza GNU Assembler para obtener el código objeto y, con la ayuda del *enlazador* GNU ld, se crea el binario final.

En la figura 2.4 se muestra un esquema general de los componentes de GCC que participan en el proceso.

El hecho de que esté dividido en estas etapas permite una compilación **modular**, es decir, cada fichero de entrada se transforma a código objeto y con la ayuda del enlazador se resuelven las dependencias que puedan existir entre ellos. A continuación, se comenta brevemente cada uno de los componentes principales.

> distcc
>
> La compilación modular y por fases permite que herramientas como `distcc` puedan realizar compilaciones distribuidas en red y en paralelo.

Preprocesador

El preprocesamiento es la primera transformación que sufre un programa en C/C++. Se lleva a cabo por el GNU CPP y, entre otras, realiza las siguientes tareas:

- Inclusión efectiva del código incluido por la directiva `#include`.
- Resuelve de las directivas `#ifdef`/`#ifndef` para la compilación condicional.
- Sustitución efectiva de todas las directivas de tipo `#define`.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Preprocesar con GCC

La opción de -E de GCC detiene la compilación justo después del preprocesamiento.

El preprocesador se puede invocar directamente utilizando la orden cpp. Como ejercicio se reserva al lector observar qué ocurre al invocar al preprocesador con el siguiente fragmento de código. ¿Se realizan comprobaciones lexícas, sintáticas o semánticas?. Utilice los parámetros que ofrece el programa para definir la macro DEFINED_IT.

Listado 2.1: Código de ejemplo preprocesable

```
1  #include <iostream>
2  #define SAY_HELLO "Hi, world!"
3
4  #ifdef DEFINED_IT
5  #warning "If you see this message, you DEFINED_IT"
6  #endif
7
8  using namespace std;
9
10 Code here??
11
12 int main() {
13   cout << SAY_HELLO << endl;
14   return 0;
15 }
```

Compilación

El código fuente, una vez preprocesado, se compila a lenguaje ensamblador, es decir, a una representación de bajo nivel del código fuente. Originalmente, la sintaxis de este lenguaje es la de AT&T pero desde algunas versiones recientes también se soporta la sintaxis de Intel.

Entre otras muchas operaciones, en la compilación se realizan las siguientes operaciones:

- Análisis **sintáctico** y **semántico** del programa. Pueden ser configurados para obtener diferentes mensajes de advertencia (*warnings*) a diferentes niveles.
- Comprobación y resolución de símbolos y dependencias a nivel de **declaración**.
- Realizar optimizaciones.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Utilizando GCC con la opción `-S` puede detenerse el proceso de compilación hasta la generación del código ensamblador. Como ejercicio, se propone cambiar el código fuente anterior de forma que se pueda construir el correspondiente en ensamblador.

GCC proporciona diferentes niveles de optimizaciones (opción `-O`). Cuanto mayor es el nivel de optimización del código resultante, mayor es el tiempo de compilación pero suele hacer más eficiente el código de salida. Por ello, se recomienda **no optimizar** el código durante las fases de desarrollo y sólo hacerlo en la fase de distribución/instalación del software.

Ensamblador

Una vez se ha obtenido el código ensamblador, GNU Assembler es el encargado de realizar la traducción a **código objeto** de cada uno de los módulos del programa. Por defecto, el código objeto se genera en archivos con extensión `.o` y la opción `-c` de GCC permite detener el proceso de compilación en este punto.

GNU Assembler

GNU Assembler forma parte de la distribución GNU Binutils y se corresponde con el programa `as`.

Como ejercicio, se propone al lector modificar el código ensamblador obtenido en la fase anterior sustituyendo el mensaje original `"Hi, world"` por `"Hola, mundo"`. Generar el código objeto asociado utilizando directamente el ensamblador (no GCC).

Enlazador

Con todos los archivos objetos el **enlazador** (*linker*) es capaz de generar el ejecutable o código binario final. Algunas de las tareas que se realizan en el proceso de enlazado son las siguientes:

- Selección y filtrado de los objetos necesarios para la generación del binario.
- Comprobación y resolución de símbolos y dependencias a nivel de **definición**.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

- Realización del enlazado (estático y dinámico) de las bibliotecas.

Como ejercicio se propone utilizar el *linker* directamente con el código objeto generado en el apartado anterior. Nótese que las opciones `-l` y `-L` sirven para añadir rutas personalizadas a las que por defecto `ld` utiliza para buscar bibliotecas.

> **GNU Linker**
>
> GNU Linker también forma parte de la distribución GNU Binutils y se corresponde con el programa `ld`.

Ejemplo de compilación con GCC

Como se ha mostrado, el proceso de compilación está compuesto por varias fases bien diferenciadas. Sin embargo, con GCC se integra todo este proceso de forma que, a partir del código fuente se genere el binario final.

En esta sección se mostrarán ejemplos en los que se crea un ejecutable al que, posteriormente, se enlaza con una biblioteca estática y otra dinámica.

Compilación de un ejecutable

Como ejemplo de ejecutable se toma el siguiente programa:

Listado 2.2: Programa básico de ejemplo.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Square {
6  private:
7    int side_;
8
9  public:
10   Square(int side_length) : side_(side_length) { };
11   int getArea() const { return side_*side_; };
12 };
13
14 int main () {
15   Square square(5);
16   cout << "Area: " << square.getArea() << endl;
17   return 0;
18 }
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

En C++, los programas que generan un ejecutable deben tener definida la función `main`, que será el punto de entrada de la ejecución. El programa es trivial: se define una clase `Square` que representa a un cuadrado. Ésta implementa un método `getArea()` que devuelve el área del cuadrado.

Suponiendo que el archivo que contiene el código fuente se llama `main.cpp`, para construir el binario utilizaremos `g++`, el compilador de C++ que se incluye en GCC. Se podría utilizar `gcc` y que se seleccionara automáticamente el compilador. Sin embargo, es una buena práctica utilizar el compilador correcto:

```
$ g++ -o main main.cpp
```

Nótese que todo el proceso de compilación se ha realizado automáticamente y cada una de las herramientas auxiliares se han ejecutado internamente en su momento oportuno (preprocesamiento, compilación, ensamblado y enlazado).

La opción `-o` indica a GCC el nombre del archivo de salida de la compilación.

Compilación de un ejecutable (modular)

En un proyecto, lo natural es dividir el código fuente en módulos que realizan operaciones concretas y bien definidas. En el ejemplo, podemos considerar un módulo la declaración y definición de la clase **Square**. Esta extracción se puede realizar de muchas maneras. Lo habitual es crear un fichero de cabecera `.h` con la declaración de la clase y un fichero `.cpp` con la definición:

Listado 2.3: Archivo de cabecera `Square.h`.

```
1 class Square {
2 private:
3   int side_;
4
5 public:
6   Square(int side_length);
7   int getArea() const;
8 };
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Listado 2.4: Implementación (Square.cpp).

```
1  #include "Square.h"
2
3  Square::Square (int side_length) : side_(side_length)
4  { }
5
6  int
7  Square::getArea() const
8  {
9    return side_*side_;
10 }
```

De esta forma, el archivo main.cpp quedaría como sigue:

Listado 2.5: Programa principal.

```
1  #include <iostream>
2  #include "Square.h"
3
4  using namespace std;
5
6  int main () {
7    Square square(5);
8    cout << "Area: " << square.getArea() << endl;
9    return 0;
10 }
```

Para construir el programa, se debe primero construir el código objeto del módulo y añadirlo a la compilación de la función principal main. Suponiendo que el archivo de cabecera se encuentra en un directorio llamado headers, la compilación puede realizarse de la siguiente manera:

```
$ g++ -Iheaders -c Square.cpp
$ g++ -Iheaders -c main.cpp
$ g++ Square.o main.o -o main
```

También se puede realizar todos los pasos al mismo tiempo:

```
$ g++ -Iheaders Square.cpp main.cpp -o main
```

Con la opción -I, que puede aparecer tantas veces como sea necesario, se puede añadir rutas donde se buscarán las cabeceras. Nótese que, por ejemplo, en main.cpp se incluyen las cabeceras usando los símbolos <> y . Se recomienda utilizar los primeros para el caso en que las cabeceras forman parte de una API pública (si existe) y deban ser utilizadas por otros programas. Por su parte, las comillas se suelen utilizar para cabeceras internas al proyecto. Las rutas por defecto son el directorio actual . para las cabeceras incluidas con y para el resto el

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

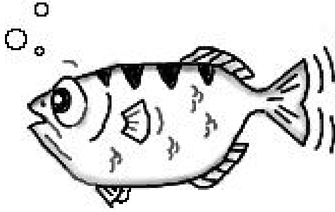

Figura 2.5: GDB es uno de los depuradores más utilizados.

directorio del sistema (normalmente, /usr/include).

Como norma general, una buena costumbre es generar todos los archivos de código objeto de un módulo y añadirlos a la compilación con el programa principal.

2.1.4. Depuración

Los programas tienen fallos y los programadores cometen errores. Los compiladores ayudan a la hora de detectar errores léxicos, sintácticos y semánticos del lenguaje de entrada. Sin embargo, el compilador no puede deducir (por lo menos hasta hoy) la *lógica* que encierra el programa, su significado final o su propósito. Estos errores se conocen como **errores lógicos**.

Los **depuradores** son programas que facilitan la labor de detección de errores, sobre todo los lógicos. Con un depurador, el programador puede probar una ejecución paso por paso, examinar/modificar el contenido de las variables en un cierto momento, etc. En general, se pueden realizar las tareas necesarias para conseguir reproducir y localizar un error difícil de detectar a simple vista. Muchos entornos de desarrollo como Eclipse, .NET o Java Beans incorporan un depurador para los lenguajes soportados. Sin duda alguna, se trata de una herramienta esencial en cualquier proceso de desarrollo software.

A continuación, se muestra el uso básico de GNU Debugger (GDB), un depurador libre para sistemas GNU/Linux que soporta diferentes lenguajes de programación, entre ellos C++.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

2.1.5. Caso de estudio: GNU GDB

Compilar para depurar

GDB necesita información extra que, por defecto, GCC no proporciona para poder realizar las tareas de depuración. Para ello, el código fuente debe ser compilado con la opción -ggdb. Todo el código objeto debe ser compilado con esta opción de compilación, por ejemplo:

```
$ g++ -Iheaders -ggdb -c module.cpp
$ g++ -Iheaders -ggdb module.o main.cpp -o main
```

Para depurar **no** se debe hacer uso de las optimizaciones. Éstas pueden generar código que nada tenga que ver con el original.

Arrancando una sesión GDB

Como ejemplo, se verá el siguiente fragmento de código:

Listado 2.6: main.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Test {
6    int _value;
7  public:
8    void setValue(int a) { _value = a; }
9    int getValue() { return _value; }
10 };
11
12 float functionB(string str1, Test* t) {
13   cout << "Function B: " << str1 << ", " << t->getValue() << endl;
14   return 3.14;
15 }
16
17 int functionA(int a) {
18   cout << "Function A: " << a << endl;
19   Test* test = NULL; /** ouch! **/
20   test->setValue(15);
21   cout << "Return B: " << functionB("Hi", test) << endl;
22   return 5;
23 }
24
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

```
25 int main() {
26   cout << "Main start" << endl;
27   cout << "Return A: " << functionA(24) << endl;
28   return 0;
29 }
```

La orden para generar el binario con símbolos de depuración sería:

```
$ g++ -ggdb main.cpp -o main
```

Si se ejecuta el código se obtienes la siguiente salida:

```
$ ./main
Main start
Function A: 24
Segmentation fault
```

Una violación de segmento (*segmentation fault*) es uno de los errores lógicos típicos de los lenguajes como C++. El problema es que se está accediendo a una zona de la memoria que no ha sido reservada para el programa, por lo que el sistema operativo interviene denegando ese acceso indebido.

A continuación, se muestra cómo iniciar una sesión de depuración con GDB para encontrar el origen del problema:

```
$ gdb main
...
Reading symbols from ./main done.
(gdb)
```

Como se puede ver, GDB ha cargado el programa, junto con los símbolos de depuración necesarios, y ahora se ha abierto una línea de órdenes donde el usuario puede especificar sus acciones.

Con los programas que fallan en tiempo de ejecución se puede generar un archivo de *core*, es decir, un fichero que contiene un volcado de la memoria en el momento en que ocurrió el fallo. Este archivo puede ser cargado en una sesión de GDB para ser examinado usando la opción `-c`.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Examinando el contenido

Para comenzar la ejecución del programa se puede utilizar la orden start:

```
(gdb) start
Temporary breakpoint 1 at 0x400d31: file main.cpp, line 26.
Starting program: main

Temporary breakpoint 1, main () at main.cpp:26
26      cout << "Main start" << endl;
(gdb)
```

De esta forma, se ha comenzado la ejecución del programa y se ha detenido en la primera instrucción de la función main(). Para reiniciar la ejecución basta con volver a ejecutar start.

Para ver más en detalle sobre el código fuente se puede utilizar la orden list o símplemente l:

```
(gdb) list
21     cout << "Return B: " << functionB("Hi", test) << endl;
22     return 5;
23   }
24
25   int main() {
26     cout << "Main start" << endl;
27     cout << "Return A: " << functionA(24) << endl;
28     return 0;
29   }
(gdb)
```

Como el resto de órdenes, list acepta parámetros que permiten ajustar su comportamiento.

> **Abreviatura**
>
> Todas las órdenes de GDB pueden escribirse utilizando su abreviatura. Ej: run = r.

Las órdenes que permiten realizar una ejecución controlada son las siguientes:

- step (s): ejecuta la instrucción actual y salta a la inmediatamente siguiente sin mantener el nivel de la ejecución (*stack frame*), es decir, entra en la definición de la función (si la hubiere).

  stepi se comporta igual que step pero a nivel de instrucciones máquina.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

- `next` (n): ejecuta la instrucción actual y salta a la siguiente manteniendo el *stack frame*, es decir, la definición de la función se toma como una instrucción atómica.

  `nexti` se comporta igual que `next` pero si la instrucción es una llamada a función se espera a que termine.

A continuación se va a utilizar `step` para avanzar en la ejecución del programa. Nótese que para repetir la ejecución de `step` basta con introducir una orden vacía. En este caso, GDB vuelve a ejecutar la última orden.

```
(gdb) s
Main start
27     cout << "Return A: " << functionA(24) << endl;
(gdb)
functionA (a=24) at main.cpp:18
18     cout << "Function A: " << a << endl;
(gdb)
```

En este punto se puede hacer uso de las órdenes para mostrar el contenido del parámetro `a` de la función `functionA()`:

```
(gdb) print a
$1 = 24
(gdb) print &a
$2 = (int *) 0x7fffffffe1bc
```

Con `print` y el modificador `&` se puede obtener el contenido y la dirección de memoria de la variable, respectivamente. Con `display` se puede configurar GDB para que muestre su contenido en cada paso de ejecución.

También es posible cambiar el valor de la variable `a`:

```
gdb) set variable a=8080
gdb) print a
3 = 8080
gdb) step
unction A: 8080
9      Test* test = NULL; /** ouch! **/
gdb)
```

La ejecución está detenida en la línea 19 donde un comentario nos avisa del error. Se está creando un puntero con el valor NULL. Posteriormente, se invoca un método sobre un objeto que no está convenientemente inicializado, lo que provoca la violación de segmento:

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

```
(gdb) next
20     test->setValue(15);
(gdb)

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400df2 in Test::setValue (this=0x0, a=15) at main.cpp:8
8      void setValue(int a) { _value = a; }
```

Para arreglar este fallo el basta con construir convenientemente el objeto:

Listado 2.7: functionA arreglada

```
17 int functionA(int a) {
18   cout << "Function A: " << a << endl;
19   Test* test = new Test();
20   test->setValue(15);
21   cout << "Return B: " << functionB("Hi", test) << endl;
22   return 5;
23 }
```

Breakpoints

La ejecución paso a paso es una herramienta útil para una depuración de grano fino. Sin embargo, si el programa realiza grandes iteraciones en bucles o es demasiado grande, puede ser un poco incómodo (o inviable). Si se tiene la sospecha sobre el lugar donde está el problema se pueden utilizar puntos de ruptura o *breakpoints* que permite detener el flujo del programa en un punto determinado por el usuario.

Con el ejemplo ya arreglado, se configura un breakpoint en la función `functionB()` y otro en la línea 28 con la orden `break`. A continuación, se ejecuta el programa hasta que se alcance el breakpoint con la orden `run` (`r`):

```
(gdb) break functionB
Breakpoint 1 at 0x400c15: file main.cpp, line 13.
(gdb) break main.cpp:28
Breakpoint 2 at 0x400ddb: file main.cpp, line 28.
(gdb) run
Starting program: main
Main start
Function A: 24

Breakpoint 1, functionB (str1=..., t=0x602010) at gdb-fix.cpp:13
13     cout << "Function B: " << str1 << ", " << t->getValue() << endl;
(gdb)
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

¡No hace falta escribir todo!. Utiliza `TAB` para completar los argumentos de una orden.

Con la orden `continue` (`c`) la ejecución avanza hasta el siguiente punto de ruptura (o fin del programa):

```
(gdb) continue
Continuing.
Function B: Hi, 15
Return B: 3.14
Return A: 5

Breakpoint 2, main () at gdb-fix.cpp:28
28    return 0;
(gdb)
```

Los breakpoint pueden habilitarse, inhabilitarse y/o eliminarse en tiempo de ejecución. Además, GDB ofrece un par de estructuras similares útiles para otras situaciones:

- **Watchpoints**: la ejecución se detiene cuando una determinada expresión cambia.
- **Catchpoints**: la ejecución se detiene cuando se produce un evento, como una excepción o la carga de una librería dinámica.

Stack y frames

En muchas ocasiones, los errores vienen debidos a que las llamadas a funciones no se realizan con los parámetros adecuados. Es común pasar punteros no inicializados o valores incorrectos a una función/método y, por tanto, obtener un error lógico.

Para gestionar las llamadas a funciones y procedimientos, en C/C++ se utiliza la pila (*stack*). En la pila se almacenan *frames*, estructuras de datos que registran las variables creadas dentro de una función así como otra información de contexto. GDB permite manipular la pila y los frames de forma que sea posible identificar un uso indebido de las funciones.

Con la ejecución parada en `functionB()`, se puede mostrar el contenido de la pila con la orden `backtrace` (`bt`):

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

```
(gdb) backtrace
#0  functionB (str1=..., t=0x602010) at main.cpp:13
#1  0x0000000000400d07 in functionA (a=24) at main.cpp:21
#2  0x0000000000400db3 in main () at main.cpp:27
(gdb)
```

Con `up` y `down` se puede navegar por los frames de la pila, y con `frame` se puede seleccionar uno en concreto:

```
(gdb) up
#1  0x0000000000400d07 in functionA (a=24) at gdb-fix.cpp:21
21    cout << "Return B: " << functionB("Hi", test) << endl;
(gdb)
#2  0x0000000000400db3 in main () at gdb-fix.cpp:27
27    cout << "Return A: " << functionA(24) << endl;
(gdb) frame 0
#0  functionB (str1=..., t=0x602010) at gdb-fix.cpp:13
13    cout << "Function B: " << str1 << ", " << t->getValue() << endl;
(gdb)
```

Invocar funciones

La orden `call` se puede utilizar para invocar funciones y métodos .

Una vez seleccionado un frame, se puede obtener toda la información del mismo, además de modificar las variables y argumentos:

```
(gdb) print *t
$1 = {_value = 15}
(gdb) call t->setValue(1000)
(gdb) print *t
$2 = {_value = 1000}
(gdb)
```

Entornos gráficos para GDB

El aprendizaje de GDB no es sencillo. La interfaz de línea de órdenes es muy potente pero puede ser difícil de asimilar, sobre todo en los primeros pasos del aprendizaje de la herramienta. Por ello, existen diferentes versiones gráficas que, en definitiva, hacen más accesible el uso de GDB:

- **GDB TUI**: normalmente, la distribución de GDB incorpora una interfaz basada en modo texto accesible pulsando `Ctrl` + `x` y, a continuación, `a`.
- **ddd** y **xxgdb**: las librerías gráficas utilizadas son algo anticuadas, pero facilitan el uso de GDB.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

- **gdb-mode**: modo de Emacs para GDB. Dentro del modo se puede activar la opción `M-x many-windows` para obtener buffers con toda la información disponible.
- **kdbg**: más atractivo gráficamente (para escritorios KDE).

2.2. Gestión de proyectos

Los proyectos software pueden ser realizados por varios equipos de personas, con formación y conocimientos diferentes, que deben *colaborar* entre sí. Además, una componente importante del proyecto es la *documentación* que se genera durante el transcurso del mismo, es decir, cualquier documento escrito o gráfico que permita entender mejor los componentes del mismo y, por ello, asegure una mayor mantenibilidad para el futuro (manuales, diagramas, especificaciones, etc.).

La *gestión del proyecto* es un proceso transversal al resto de procesos y tareas y se ocupa de la planificación y asignación de los recursos de los que se disponen. Se trata de un proceso *dinámico*, ya que debe adaptarse a los diferentes cambios e imprevistos que puedan surgir durante el desarrollo. Para detectar estos cambios a tiempo, dentro de la gestión del proyecto se realizan *tareas de seguimiento* que consisten en registrar y notificar errores y retrasos en las diferentes fases y entregas.

Existen muchas herramientas que permiten crear **entornos colaborativos** que facilitan el trabajo tanto a desarrolladores como a los jefes de proyecto, los cuales están más centrados en las tareas de gestión. En esta sección se presentan algunos entornos colaborativos actuales, así como soluciones específicas para un proceso concreto.

2.2.1. Sistemas de control de versiones

El resultado más importante de un proyecto software son los archivos de distinto tipo que se generan; desde el código fuente, hasta los diseños, bocetos y documentación del mismo. Desde el punto de vista técnico, se trata de gestionar una cantidad importante de archivos que son modificados a lo largo del tiempo por diferentes personas. Además, es posible que para un conjunto de archivos sea necesario volver a una versión anterior.

Por ejemplo, un fallo de diseño puede implicar un código que no es escalable. Si es posible volver a un estado original conocido, el tiempo invertido en *revertir los cambios* es menor.

Por otro lado, sería interesante tener la posibilidad de realizar *desarrollos en paralelo* de forma que exista una versión «estable» de to-

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

do el proyecto y otra más «experimental» del mismo donde se probaran diferentes algoritmos y diseños. De esta forma, probar el impacto que tendría nuevas implementaciones sobre el proyecto no afectaría a una versión más «oficial». También es común que se desee añadir una nueva funcionalidad y ésta se realiza en paralelo junto con otros desarrollos.

Los **sistemas de control de versiones** o Version Control System (VCS) permiten gestionar los archivos de un proyecto (y sus versiones) y que sus integrantes puedan acceder remotamente a ellos para descargarlos, modificarlos y publicar los cambios. También se encargan de detectar posibles conflictos cuando varios usuarios modifican los mismos archivos y de proporcionar un sistema básico de registro de cambios.

Como norma general, al VCS debe subirse el archivo fuente y nunca el archivo generado. No se deben subir binarios ya que no es fácil seguir la pista a sus modificaciones.

Sistemas centralizados vs. distribuidos

Existen diferentes criterios para clasificar los diferentes VCS existentes. Uno de los que más influye tanto en la organización y uso del repositorio es si se trata de VCS centralizado o distribuido. En la figura 2.6 se muestra un esquema de ambas filosofías.

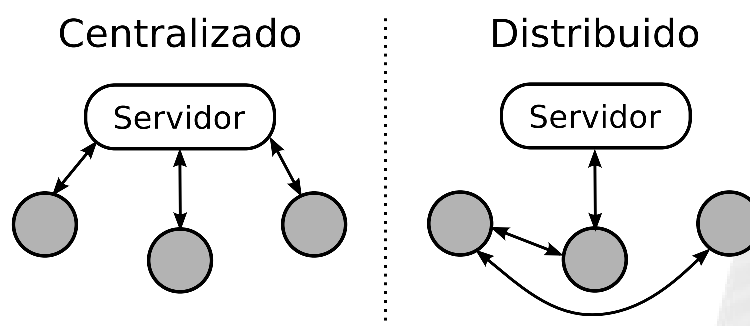

Figura 2.6: Esquema centralizado vs. distribuido de VCS.

Los VCS centralizados como CVS o *Subversion* se basan en que existe un nodo servidor con el que todos los clientes conectan para obtener los archivos, subir modificaciones, etc. La principal ventaja de este esquema reside en su sencillez: las diferentes versiones del proyecto están únicamente en el servidor central, por lo que los posibles conflictos entre las modificaciones de los clientes pueden detectarse y gestionarse

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Programación multimedia e xogos
David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

Figura 2.7: Logotipo del proyecto Apache Subversion.

más fácilmente. Sin embargo, el servidor es un único punto de fallo y en caso de caída, los clientes quedan aislados.

Por su parte, en los VCS distribuidos como *Mercurial* o *Git*, cada cliente tiene un repositorio local al nodo en el que se suben los diferentes cambios. Los cambios pueden agruparse en *changesets*, lo que permite una gestión más ordenada. Los clientes actúan de servidores para el resto de los componentes del sistema, es decir, un cliente puede descargarse una versión concreta de otro cliente.

Esta arquitectura es tolerante a fallos y permite a los clientes realizar cambios sin necesidad de conexión. Posteriormente, pueden sincronizarse con el resto. Aún así, un VCS distribuido puede utilizarse como uno centralizado si se fija un nodo como servidor, pero se perderían algunas posibilidades que este esquema ofrece.

Subversion

Subversion (SVN) es uno de los VCS centralizado más utilizado, probablemente, debido a su sencillez en el uso. Básicamente, los clientes tienen accesible en un servidor todo el repositorio y es ahí donde se envían los cambios.

Para crear un repositorio, en el servidor se puede ejecutar la siguiente orden:

```
$ svnadmin create /var/repo/myproject
```

Esto creará un árbol de directorios en `/var/repo/myproject` que contiene toda la información necesaria. Una vez hecho esto es necesario hacer accesible este directorio a los clientes. En lo que sigue, se supone que los usuarios tienen acceso a la máquina a través de una cuenta SSH, aunque se pueden utilizar otros métodos de acceso.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Es recomendable que el acceso al repositorio esté controlado. HTTPS o SSH son buenas opciones de métodos de acceso.

Inicialmente, los clientes pueden descargarse el repositorio por primera vez utilizando la orden `checkout`:

```
$ svn checkout svn+ssh://user1@myserver:/var/repo/myproject
Checked out revision X.
```

Esto ha creado un directorio `myproject` con el contenido del repositorio. Una vez descargado, se pueden realizar todos los cambios que se deseen y, posteriormente, subirlos al repositorio. Por ejemplo, añadir archivos y/o directorios:

```
$ mkdir doc
$ echo "This is a new file" > doc/new_file
$ echo "Other file" > other_file
$ svn add doc other_file
A         doc
A         doc/new_file
A         other_file
```

La operación `add` indica qué archivos y directorios han sido seleccionados para ser añadidos al repositorio (marcados con `A`). Esta operación **no** sube efectivamente los archivos al servidor. Para subir cualquier cambio se debe hacer un *commit*:

```
$ svn commit
```

A coninuación, se lanza un editor de texto[1] para que se especifique un mensaje que describa los cambios realizados. Además, también incluye un resumen de todas las operaciones que se van a llevar a cabo en el commit (en este caso, sólo se añaden elementos). Una vez terminada la edición, se salva y se sale del editor y la carga comenzará.

Cada commit aumenta en 1 el número de revisión. Ese número será el que podremos utilizar para volver a versiones anteriores del proyecto utilizando:

```
$ svn update -r REVISION
```

Si no se especifica la opción `-r`, la operación `update` trae la última revisión (*head*).

[1] El configurado en la variable de entorno EDITOR.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

En caso de que otro usuario haya modificado los mismos archivos y lo haya subido antes al repositorio central, al hacerse el commit se detectará un *conflicto*. Como ejemplo, supóngase que el cliente `user2` ejecuta lo siguiente:

```
$ svn checkout svn+ssh://user2@myserver:/var/repo/myproject
$ echo "I change this file" > other_file
$ svn commit
Committed revision X+1.
```

Y que el cliente `user1`, que está en la versión `X`, ejecuta lo siguiente:

```
$ echo "I change the content" > doc/new_file
$ svn remove other_file
D         other_file
$ svn commit
svn: Commit failed (details follow):
svn: File 'other_file' is out of date
```

Para resolver el conflicto, el cliente `user1` debe actualizar su versión:

```
$ svn update
C other_file
At revision X+1.
```

La marca `C` indica que `other_file` queda en conflicto y que debe resolverse manualmente. Para resolverlo, se debe editar el archivo donde Subversion marca las diferencias con los símbolos `'<'` y `'>'`.

También es posible tomar como solución el revertir los cambios realizados por `user1` y, de este modo, aceptar los de `user2`:

```
$ svn revert other_file
$ svn commit
Committed revision X+2.
```

Nótese que este commit sólo añade los cambios hechos en `new_file`, aceptando los cambios en `other_file` que hizo `user2`.

Mercurial

Como se ha visto, en los VCS centralizados como Subversion no se permite, por ejemplo, que los clientes hagan commits si no están conectados con el servidor central. Los VCS como Mercurial (HG), permiten que los clientes tengan un repositorio local, con su versión modificada del proyecto y la sincronización del mismo con otros servidores (que pueden ser también clientes).

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Figura 2.8: Logotipo del proyecto Mercurial.

Para crear un repositorio Mercurial, se debe ejecutar lo siguiente:

```
$ hg init /home/user1/myproyect
```

Al igual que ocurre con Subversion, este directorio debe ser accesible mediante algún mecanismo (preferiblemente, que sea seguro) para que el resto de usuarios pueda acceder. Sin embargo, el usuario `user1` puede trabajar directamente sobre ese directorio.

Para obtener una versión inicial, otro usuario (`user2`) debe *clonar* el repositorio. Basta con ejecutar lo siguiente:

```
$ hg clone ssh://user2@remote-host//home/user1/myproyect
```

A partir de este instante, `user2` tiene una versión inicial del proyecto extraída a partir de la del usuario `user1`. De forma muy similar a Subversion, con la orden `add` se pueden añadir archivos y directorios.

Mientras que en el modelo de Subversion, los clientes hacen `commit` y `update` para subir cambios y obtener la última versión, respectivamente; en Mercurial es algo más complejo, ya que existe un *repositorio local*. Como se muestra en la figura 2.9, la operación `commit` (3) sube los cambios a un repositorio local que cada cliente tiene.

Cada commit se considera un *changeset*, es decir, un conjunto de cambios agrupados por un mismo ID de revisión. Como en el caso de Subversion, en cada commit se pedirá una breve descripción de lo que se ha modificado.

Una vez hecho todos commits, para llevar estos cambios a un servidor remoto se debe ejecutar la orden de `push` (4). Siguiendo con el ejemplo, el cliente `user2` lo enviará por defecto al repositorio del que hizo la operación `clone`.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

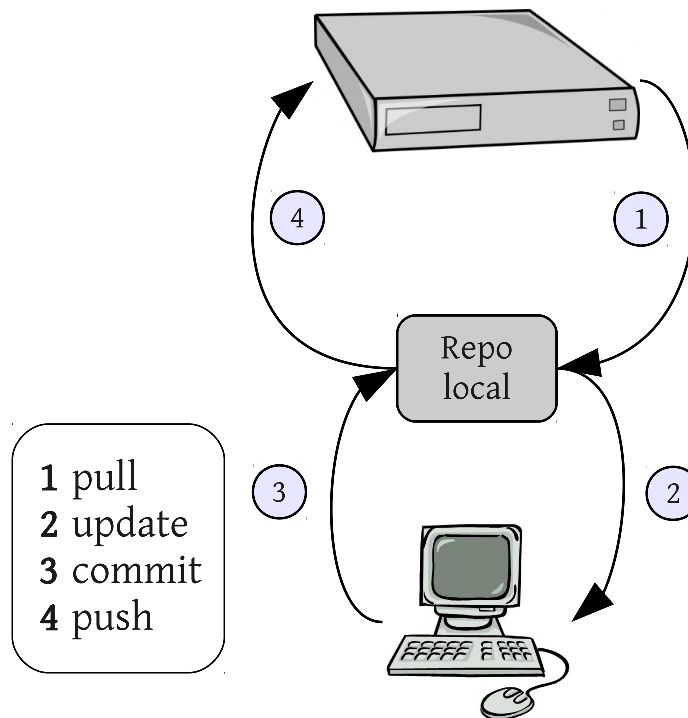

Figura 2.9: Flujo de trabajo de Mercurial

El sentido inverso, es decir, traerse los cambios del servidor remoto a la copia local, se realiza también en 2 pasos: `pull` (1) que trae los cambios del repositorio remoto al repositorio local; y `update` (2), que aplica dichos cambios del repositorio local al directorio de trabajo. Para hacer los dos pasos al mismo tiempo, se puede hacer lo siguiente:

```
$ hg pull -u
```

Para evitar conflictos con otros usuarios, una buena costumbre **antes** de realizar un `push` es conveniente obtener los posibles cambios en el servidor con `pull` y `update`.

Para ver cómo se gestionan los conflictos en Mercurial, supóngase que `user1` realiza lo siguiente:

```
$ echo "A file" > a_file
$ hg add a_file
$ hg commit
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Al mismo tiempo, `user2` ejecuta lo siguiente:

```
$ echo "This is one file" > a_file
$ hg add a_file
$ hg commit
$ hg push
abort: push creates new remote head xxxxxx!
(you should pull and merge or use push -f to force)
```

Al intentar realizar el `push` y entrar en conflicto, Mercurial avisa de ello deteniendo la carga. En este punto se puede utilizar la opción `-f` para forzar la operación de `push`, lo cual crearía un nuevo *head*. Como resultado, se crearía una nueva rama a partir de ese conflicto de forma que se podría seguir desarrollando omitiendo el conflicto. Si en el futuro se pretende unir los dos heads se utilizan las operaciones `merge` y `resolve`.

La otra solución, y normalmente la más común, es obtener los cambios con `pull`, unificar heads (`merge`), resolver los posibles conflictos manualmente si es necesario (`resolve`), hacer commit de la solución dada (`commit`) y volver a intentar la subida (`push`):

> `hgview`
>
> `hgview` es una herramienta gráfica que permite visualizar la evolución de las ramas y heads de un proyecto que usa Mercurial.

```
$ hg pull
adding changesets
adding manifests
adding file changes
added 2 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
$ hg merge
merging a_file
warning: conflicts during merge.
merging a_file failed!
0 files updated, 0 files merged, 0 files removed, 1 files unresolved
$ hg resolve -a
$ hg commit
$ hg push
```

Para realizar cómodamente la tarea de resolver los conflictos manualmente existen herramientas como `meld` que son invocadas automáticamente por Mercurial cuando se encuentran conflictos de este tipo.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

2.2.2. Documentación

Uno de los elementos más importantes que se generan en un proyecto es la documentación: cualquier elemento que permita entender mejor tanto el proyecto en su totalidad como sus partes, de forma que facilite el proceso de **mantenimiento** en el futuro. Además, una buena documentación hará más sencilla la **reutilización** de componentes.

Existen muchos formatos de documentación que pueden servir para un proyecto software. Sin embargo, muchos de ellos, tales como PDF, ODT, DOC, etc., son formatos «binarios» por lo que no son aconsejables para utilizarlos en un VCS. Además, utilizando texto plano es más sencillo crear programas que automaticen la generación de documentación, de forma que se ahorre tiempo en este proceso.

Por ello, aquí se describen algunas formas de crear documentación basadas en texto plano. Obviamente, existen muchas otras y, seguramente, sean tan válidas como las que se proponen aquí.

Doxygen

Doxygen es un sistema que permite generar la documentación utilizando analizadores de código que averiguan la estructura de módulos y clases, así como las funciones y los métodos utilizados. Además, se pueden realizar anotaciones en los comentarios del código que sirven para añadir información más detallada. La principal ventaja es que se vale del propio código fuente para generar la documentación. Además, si se añaden comentarios en un formato determinado, es posible ampliar la documentación generada con notas y aclaraciones sobre las estructuras y funciones utilizadas.

Algunos piensan que el uso de programas como Doxygen es bueno porque «obliga» a comentar el código. Otros piensan que no es así ya que los comentarios deben seguir un determinado formato, dejando de ser comentarios propiamente dichos.

El siguiente fragmento de código muestra una clase en C++ documentada con el formato de Doxygen:

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Listado 2.8: Clase con comentarios Doxygen

```
1  /**
2     This is a test class to show Doxygen format documentation.
3    */
4
5  class Test {
6  public:
7    /// The Test constructor.
8    /**
9       \param s the name of the Test.
10   */
11   Test(string s);
12
13   /// Start running the test.
14   /**
15      \param max maximum time of test delay.
16      \param silent if true, do not provide output.
17      \sa Test()
18   */
19   int run(int max, bool silent);
20 };
```

Por defecto, Doxygen genera la documentación en HTML y basta con ejecutar la siguiente orden en el raíz del código fuente para obtener una primera aproximación:

```
$ doxygen .
```

reStructuredText

reStructuredText (RST) es un formato de texto básico que permite escribir texto plano añadiendo pequeñas anotaciones de formato de forma que no se pierda legibilidad. Existen muchos traductores de RST a otros formatos como PDF (`rst2pdf`) y HTML (`rst2html`), que además permiten modificar el estilo de los documentos generados.

El formato RST es similar a la sintaxis de los sistema tipo wiki. Un ejemplo de archivo en RST puede ser el siguiente:

Listado 2.9: Archivo en RST

```
1 ================
2  The main title
3 ================
4
5 This is an example of document in ReStructured Text (RST). You can
      get
6 more info about RST format at `RST Reference
7 <http://
      docutils.sourceforge.net/docs/ref/rst/restructuredtext.html>`_.
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

```
 8
 9 Other section
10 =============
11
12 You can use bullet items:
13
14  - Item A
15
16  - Item B
17
18 And a enumerated list:
19
20  1. Number 1
21
22  2. Number 2
23
24 Tables
25 ------
26
27 +--------------+----------+-----------+-----------+
28 | row 1, col 1 | column 2 | column 3  | column 4  |
29 +--------------+----------+-----------+-----------+
30 | row 2        |          |           |           |
31 +--------------+----------+-----------+-----------+
32
33 Images
34 ------
35
36 .. image:: gnu.png
37    :scale: 80
38    :alt: A title text
```

Como se puede ver, aunque RST añade una sintaxis especial, el texto es completamente legible. Ésta es una de las ventajas de RST, el uso de etiquetas de formato que no «ensucian» demasiado el texto.

YAML

YAML (YAML Ain't Markup Language)[2] es un lenguaje diseñado para serializar datos procedentes de aplicaciones en un formato que sea legible para los humanos. Estrictamente, no se trata de un sistema para documentación, sin embargo, y debido a lo cómodo de su sintaxis, puede ser útil para exportar datos, cargar los mismos en el programa, representar configuraciones, etc. Otra de sus ventajas es que hay un gran número de bibliotecas en diferentes lenguajes (C++, Python, Java, etc.) para tratar información YAML. Las librerías permiten automáticamente salvar las estructuras de datos en formato YAML y el proceso inverso: cargar estructuras de datos a partir del YAML.

[2] http://www.yaml.org

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

En el ejemplo siguiente, extraído de la documentación oficial, se muestra una factura. De un primer vistazo, se puede ver qué campos forman parte del tipo de dato factura tales como `invoice`, `date`, etc. Cada campo puede ser de distintos tipo como numérico, booleano o cadena de caracteres, pero también listas (como `product`) o referencias a otros objetos ya declarados (como `ship-to`).

Listado 2.10: Archivo en YAML

```
1  --- !<tag:clarkevans.com,2002:invoice>
2  invoice: 34843
3  date   : 2001-01-23
4  bill-to: &id001
5      given  : Chris
6      family : Dumars
7      address:
8          lines: |
9              458 Walkman Dr.
10             Suite #292
11         city    : Royal Oak
12         state   : MI
13         postal  : 48046
14 ship-to: *id001
15
16 product:
17     - sku         : BL394D
18       quantity    : 4
19       description : Basketball
20       price       : 450.00
21     - sku         : BL4438H
22       quantity    : 1
23       description : Super Hoop
24       price       : 2392.00
25
26 tax  : 251.42
27 total: 4443.52
28
29 comments:
30     Late afternoon is best.
31     Backup contact is Nancy
32     Billsmer @ 338-4338.
```

2.2.3. Forjas de desarrollo

Hasta ahora, se han mostrado herramientas específicas que permiten crear y gestionar los elementos más importantes de un proyecto software: los archivos que lo forman y su documentación. Sin embargo, existen otras como la gestión de tareas, los mecanismos de notificación de errores o los sistemas de comunicación con los usuarios que son de utilidad en un proyecto.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Las forjas de desarrollo son sistemas colaborativos que integran no sólo herramientas básicas para la gestión de proyectos, como un VCS, sino que suelen proporcionar herramientas para:

- **Planificación y gestión de tareas**: permite anotar qué tareas quedan por hacer y los plazos de entrega. También suelen permitir asignar prioridades.
- **Planificación y gestión de recursos**: ayuda a controlar el grado de ocupación del personal de desarrollo (y otros recursos).
- **Seguimiento de fallos**: también conocido como *bug tracker*, es esencial para llevar un control sobre los errores encontrados en el programa. Normalmente, permiten gestionar el ciclo de vida de un fallo, desde que se descubre hasta que se da por solucionado.
- **Foros**: normalmente, las forjas de desarrollo permiten administrar varios foros de comunicación donde con la comunidad de usuarios del programa pueden escribir propuestas y notificar errores.

Las forjas de desarrollo suelen ser accesibles via web, de forma que sólo sea necesario un navegador para poder utilizar los diferentes servicios que ofrece. Dependiendo de la forja de desarrollo, se ofrecerán más o menos servicios. Sin embargo, los expuestos hasta ahora son los que se proporcionan habitualmente. Existen forjas gratuitas en Internet que pueden ser utilizadas para la creación de un proyecto. Algunas de ellas:

- **GNA**[3]: es una forja de desarrollo creada por la Free Software Foundation de Francia que soporta, actualmente, repositorios CVS, GNU Arch y Subversion. Los nuevos proyectos son estudiados cuidadosamente antes de ser autorizados.
- **Launchpad**[4]: forja gratuita para proyectos de software libre creada por Canonical Ltd. Se caracteriza por tener un potente sistema de *bug tracking* y proporcionar herramientas automáticas para despliegue en sistemas Debian/Ubuntu.
- **BitBucket**[5]: forja de la empresa Atlassian que ofrece repositorios Mercurial y Git. Permite crear proyectos privados gratuitos pero con límite en el número de desarrolladores por proyecto.
- **GitHub**[6]: forja proporcionada por GitHub Inc. que utiliza repositorios Git. Es gratuito siempre que el proyecto sea público, es decir,

[3] http://gna.org
[4] https://launchpad.net/
[5] http://bitbucket.org
[6] http://github.com

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

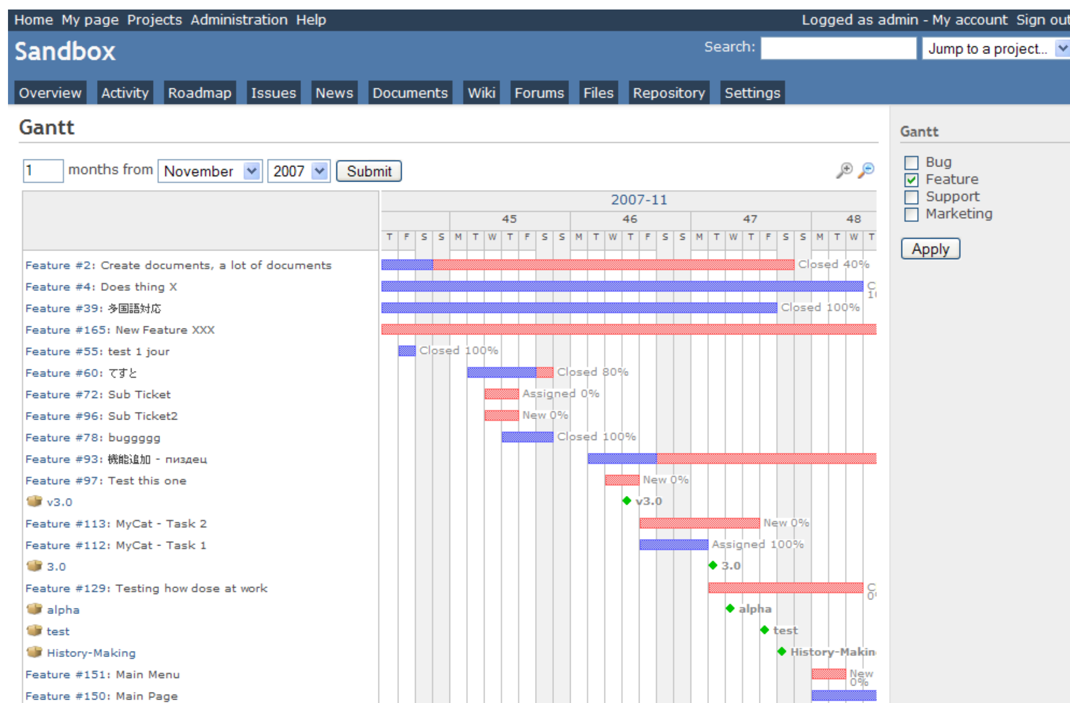

Figura 2.10: Aspecto de la herramienta de gestión de tareas de Redmine

pueda ser descargado y modificado por cualquier usuario de Github sin restricción.

- **SourceForge**[7]: probablemente, una de las forjas gratuitas más conocidas. Propiedad de la empresa GeekNet Inc., soporta Subversion, Git, Mercurial, Bazaar y CVS.
- **Google Code**[8]: la forja de desarrollo de Google que soporta Git, Mercurial y Subversion.

Redmine

Además de los servicios gratuitos presentados, existe gran variedad de software que puede ser utilizado para gestionar un proyecto de forma que pueda ser instalado en un servidor personal y así no depender de un servicio externo. Tal es el caso de Redmine (véase figura 2.10) que entre las herramientas que proporciona cabe destacar las siguientes características:

[7] http://sourceforge.net
[8] http://code.google.com

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

- Permite crear **varios proyectos**. También es configurable qué servicios se proporcionan en cada proyecto: gestor de tareas, tracking de fallos, sistema de documentación wiki, etc.
- **Integración con repositorios**, es decir, el código es accesible a través de Redmine y se pueden gestionar tareas y errores utilizando los comentarios de los commits.
- **Gestión de usuarios** que pueden utilizar el sistema y con qué políticas de acceso.
- Está construido en **Ruby** y existe una amplia variedad de **plugins** que añaden funcionalidad extra.

2.3. NME. Toma de contacto

2.3.1. ¿Qué es NME?

Descripción general

En palabras de sus creadores[9], *NME es la mejor forma de crear juegos multi-plataforma*. El optimismo excesivo puede jugar, en algunas ocasiones, malas pasadas, pero en el caso particular de NME es cierto que representa uno de los *frameworks open-source* más potentes y fáciles de utilizar existentes en la actualidad.

El hecho de que NME sea un ***framework* multi-plataforma** permite la generación de distintos ejecutables, a partir del mismo código fuente, que podrán *correr* sobre distintos plataformas, desde ordenadores personales con Windows o GNU/Linux hasta navegadores web con soporte de HTML5 o Flash pasando por teléfonos y *tablets* con Android o iOS.

> Compilación cruzada
>
> Un compilador cruzado es capaz de generar código ejecutable para otra plataforma distinta a aquélla en la que se ejecuta.

Este proceso de abstracción que permite, reutilizando prácticamente el mismo código fuente, la generación de ejecutables asociados a diversos lenguajes de programación se consigue gracias a que NME hace uso del innovador **lenguaje de programación Haxe**. De hecho, el

[9] http://www.nme.io/about

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Figura 2.11: La primera versión liberada en Internet de NME fue el 1 de Marzo de 2007.

compilador de Haxe es capaz de generar el mismo código fuente a código C++, JavaScript y *bytecode* SWF sin realizar sacrificios importantes en la fiabilidad de dicho software. Como resultado, NME proporciona un buen rendimiento para Windows, Mac, GNU/Linux, iOS, Android y BlackBerry, ofreciendo también soporte para Flash Player y HTML5.

NME proporciona un buen balance entre un desarrollo ágil con un flujo de trabajo fácil de usar y la generación de juegos o aplicaciones de calidad.

Características

NME hace gala de una serie de características que lo convierten en una elección muy interesante para el desarrollo de videojuegos. A continuación se enumeran las principales cualidades que lo caracterizan.

Desde el punto de vista **gráfico**, NME

- ofrece soporte para *bitmaps*,
- permite tratar con gráficos vectoriales,
- soporta manipulación a nivel de píxel,
- soporta *batch triangle rendering*,
- soporta *batch tile rendering*,
- permite un acceso directo a la API OpenGL para tener más control sobre el proceso de *rendering*.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

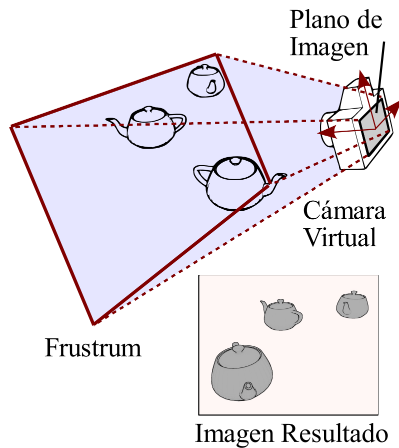

Figura 2.12: Visión general del **proceso de *rendering***, cuyo objetivo principal es la generación de imágenes 2D a partir de la descripción abstracta de una escena 3D.

- soporte para *rendering* a pantalla completa.

Desde el punto de vista **multimedia**, NME

- soporta la reproducción de audio,
- posibilita la generación dinámica de audio.

Desde el punto de vista de las redes de computadores, un *socket* representa un canal de comunicación entre procesos a través de una red. La programación de *sockets* se realiza, comúnmente, haciendo uso de los protocolos TCP o UDP.

Desde el punto de vista del **networking**, NME

- soporta peticiones HTTP,
- permite el uso de *sockets* de bajo nivel,
- proporciona soporte para cachés de datos.

Desde el punto de vista de la manipulación de **texto**, NME

- soporta el *rendering* de texto a nivel de dispositivo,
- soporta el *rendering* de fuentes empotradas,
- posibilita la personalización del *rendering* de fuentes,

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

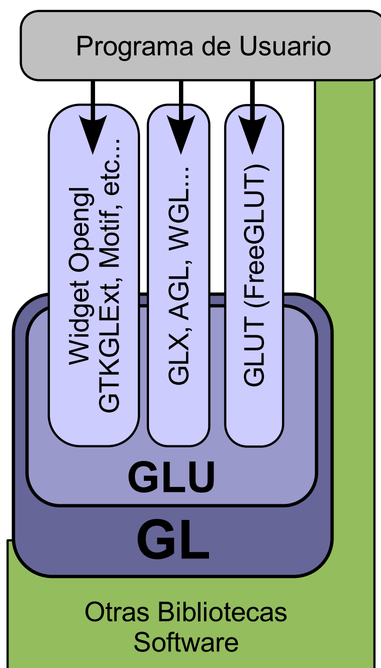

Figura 2.13: OpenGL es una API multiplataforma para el renderizado interactivo de gráficos en 2D/3D que fue inicialmente desarrollada por Silicon Graphics en 1992. El gráfico muestra la relación de la biblioteca asociada a OpenGL (GL) con otras bibliotecas relevantes en este contexto.

- ofrece *rendering* HTML básico.

Desde el punto de vista de la **gestión de eventos**, NME

- facilita el tratamiento de eventos de teclado y ratón.
- soporta entrada *multitouch*.
- permite el uso de *joysticks*.

Por otra parte, resulta importante destacar que NME permite hacer uso de **extensiones nativas** a determinados lenguaje de programación que tengan dicho soporte. Por ejemplo, NME posibilita la programación multi-hilo para plataformas que proporcionen, de manera nativa, dicho modelo de programación. De este modo, es posible mejorar la eficiencia, si así es necesario, atendiendo al dispositivo final sobre el cual se desplegará el videojuego desarrollado.

Además, NME se puede utilizar fácilmente con **otras bibliotecas** para integrar funcionalidad no trivial en muchas casos. En el contexto del desarrollo de videojuegos, existe cierta funcionalidad que es recurrente y se utiliza con mucha frecuencia, como por ejemplo las animaciones de *sprites*, la física de cuerpos rígidos o los efectos de partículas. Afortunadamente, existen bibliotecas que proporcionan esta funcionalidad (y mucha más) y que se integran fácilmente con NME.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

Figura 2.14: Un *sprite* es una imagen o animación 2D que se integra dentro de una escena mayor.

Por último, NME es un *framework* que posibilita la integración con otros *frameworks* existentes para el desarrollo de videojuegos. En este contexto, algunos ejemplos representativos son *Stencyl*[10], *Flixel*[11], *HaxePunk*[12] o *awe6*[13].

Plataformas soportadas

Como se ha introducido anteriormente, NME soporta una **gran variedad** de plataformas software:

- Windows.
- Mac.
- GNU/Linux.
- iOS.
- Android.
- BlackBerry.
- webOS.
- Flash.
- HTML5.

La consecuencia inmediata de esta característica es que el **mercado potencial** para desplegar un juego desarrollado con NME se multiplica

[10] http://www.stencyl.com
[11] http://flixel.org
[12] http://www.haxepunk.com
[13] http://awe6.org

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

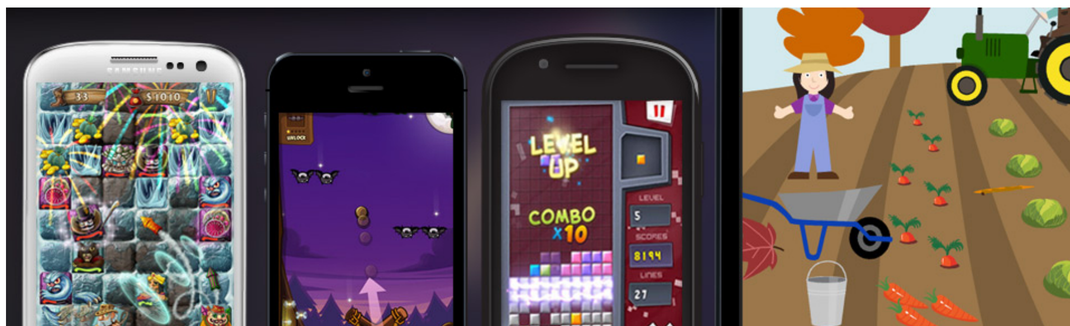

Figura 2.15: *Target Everything* es una de las señas de identidad del *framework* NME y, sin duda, uno de sus puntos fuertes. Imagen obtenida de http://www.nme.io/

por un factor más que relevante. Por ejemplo, un desarrollo de un videojuego con NME se podría ejecutar, con una mínima inversión de tiempo extra para resolver cuestiones específicas que puedan surgir, en un teléfono móvil con Android 3.0, un iPhone con iOS 5.0 y, por ejemplo, un navegador web como Chromium con soporte para HTML5.

No obstante, a veces todo no es tan directo y sencillo. Comúnmente, en los desarrollos multi-plataforma siempre hay que considerar **aspectos específicos** de las plataformas finales sobre las que se ejecutará el videojuego (o aplicación) desarrollado. Esto se debe fundamentalmente a dos factores: i) controlar que los aspectos implementación del juego están soportados por las plataformas finales y ii) incrementar la eficiencia en una determinada plataforma, lo cual implicará implementar código específico en la misma.

> *Cross-platform SW*
>
> El SW multi-plataforma se puede dividir en dos tipos: i) SW que ha de compilarse para la plataforma destino y ii) SW que se puede ejecutar directamente sobre la plataforma destino sin necesidad de una preparación especial.

Llegados a este punto, el lector podría ser reticente a pensar que NME permita la generación de código para múltiples plataformas y, que de manera simultánea, sea fácil de usar y realmente práctico para que el proceso de desarrollo multi-plataforma sea ágil. Sin embargo, y como se discutirá más adelante en la sección 2.3.3, realmente lo es. De hecho, y tras realizar con éxito la instalación de NME y su configuración inicial, generar el código en C++ de un videojuego que se ejecutará en sistemas GNU/Linux es tan sencillo como ejecutar la siguiente instrucción:

```
$ nme test "Ejemplo.nmml" linux
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Figura 2.16: La versión actual de Haxe es la 2.10. Desde que en 2005 apareciese su primera versión, la comunidad Haxe ha logrado desarrollar un lenguaje de programación multi-plataforma de alto nivel que gana adeptos cada día.

Con esta instrucción, NME es capaz de generar el código C++ a partir del implementado en Haxe y de crear un ejecutable binario que pueda ejecutarse.

A continuación se describe, desde un punto de vista general, uno de los pilares de NME que permite que todo esto sea posible: el lenguaje de programación multi-plataforma Haxe.

2.3.2. El lenguaje de programación Haxe

Descripción general

Haxe[14] es un lenguaje de programación de código abierto. Sin embargo, mientras otros lenguajes están estrechamente ligados a una plataforma, como por ejemplo Java a su máquina virtual (*Java Virtual Machine*), Haxe es un lenguaje multi-plataforma. Esto significa que Haxe puede utilizarse con el objetivo de generar código para plataformas como Javascript, Flash, NekoVM[15] (*Neko Virtual Machine*), PHP, C++, C# y Java.

Haxe representa el corazón de NME.

La filosofía de Haxe se basa en permitir que el programador decida la mejor plataforma destino para un programa. Este enfoque incrementa

[14] http://www.haxe.org

[15] Neko es un lenguaje de programación dinámicamente tipado y de alto nivel. Normalmente, se utiliza como lenguaje de *scripting*.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Figura 2.17: El soporte multi-plataforma de Haxe es muy amplio y permite mejorar drásticamente el retorno de una inversión.

de manera evidente la **flexibilidad** a la hora de desarrollar, ya que cada plataforma está típicamente asociada a un lenguaje de programación concreto. Los tres pilares sobre los que esta filosofía se fundamenta se enumeran a continuación:

1. Lenguaje de programación estandarizado.
2. Biblioteca estándar de funciones que funciona de manera idéntica en todas las plataformas.
3. Bibliotecas específicas de plataforma, posibilitando el acceso a las mismas desde código Haxe.

Antes de describir desde un punto de vista general las características más relevantes de Haxe como lenguaje de programación, resulta interesante distinguir entre dos grupos a la hora de estudiar el soporte multi-plataforma proporcionado por Haxe: i) *client-side* (*lado* del cliente) y ii) *server-side* (*lado* del servidor).

Por una parte, desde el punto de vista del **cliente de una aplicación**, Haxe proporciona la siguiente funcionalidad:

- Es posible compilar a JavaScript generando un único archivo .js. Además, se permite el *debug* interactivo directamente en código Haxe y es posible ajustar el tamaño final del script resultante.
- Es posible compilar a código C++, el cual se puede compilar a su vez a binarios nativos de plataformas móviles, como iOS o Android. De hecho, NME es un caso representativo de este enfoque.
- Es posible general a Flash generando un archivo .swf. Además, se permite el *debug* interactivo y es posible integrar bibliotecas externas SWF.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Por otra parte, desde el punto de vista del **servidor de aplicaciones**, Haxe proporciona la siguiente funcionalidad:

- Es posible compilar a NodeJS y a otras tecnologías relacionadas con JavaScript desde el punto de vista del servidor.
- Es posible compilar a código PHP 5.
- Es posible compilar para la máquina virtual de Neko generando un único archivo binario .n.

Características del lenguaje

En este apartado se enumeran las principales características del lenguaje de programación Haxe con el objetivo de que el lector pueda tener una primera impresión de lo que ofrece y lo que no ofrece Haxe.

- Modelo de **POO (Programación Orientada a Objetos)**, el cual facilita el diseño, desarrollo y mantenimiento de proyectos software con una complejidad significativa. Haxe da soporte, mediante distintos elementos, a las características típicas de la OO (Orientación a Objetos) (herencia, encapsulación y polimorfismo).

> Tipado fuerte
>
> Un lenguaje de programación se considera fuertemente tipado cuando no se puede utilizar una variable de un tipo determinado, al menos de manera directa, como si fuera de otro tipo distinto.

- Tipado estricto pero con un **soporte dinámico de tipos**. Esta característica garantiza la interoperabilidad con bibliotecas dependiente de una plataforma en concreto.
- Soporte a **paquetes y módulos** que permitan organizar el código de la manera más adecuada posible.
- Integración de **tipos genéricos** para permitir una programación genérica. Por ejemplo, la clase *Array* de Haxe se puede instanciar para albergar distintos tipos de datos en función de las necesidades de un programa.

> *Type inference*
>
> El compilador de Haxe es capaz de traducir la instrucción $var f = 1{,}0$ a $var f : Float = 1{,}0$ sin necesidad de que el desarrollador especifique el tipo de una variable.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

- **Inferencia de tipos** que permite que el compilador de Haxe *deduzca* el tipo de una variable a partir del valor que le fue asignado previamente. Esta característica libera al desarrollador de las restricciones de un lenguaje fuertemente tipado.

- Soporte a **parámetros opcionales y por defecto** en funciones, permitiendo establecer esquemas por defecto a la hora de invocar a funciones.

- Integración del modificador **inline** tanto a nivel de variable como a nivel de función. En el caso de las variables, el uso de *inline* provoca que el compilador sustituya el nombre de la misma por su valor. En el caso de las llamadas a funciones, el uso de *inline* provoca que el compilador sustituya la llamada a una función por el código que conforma su cuerpo. Este modificador es especialmente relevante para incrementar el rendimiento de una aplicación, sobre todo en el caso de variables o funciones que se utilizan con una gran frecuencia.

- Manejo de **excepciones** mediante los típicos bloques *try-catch*.

- Integración de **metadatos** para, por ejemplo, definir un comportamiento específico. Los metadatos se pueden recuperar en tiempo de ejecución.

> *getX()* y *setX()*
>
> Las operaciones *setters* (escritura) y *getters* (lectura) definen el tipo de acceso a una variable miembro de una clase.

- Integración de **propiedades** para, por ejemplo, implementar distintos tipos de características como las variables miembro accedidas mediante *setters* o *getters*.

- Soporte a **compilación condicional** a través de macros de compilación condicional. Este esquema facilita la integración de código específico para una determinada plataforma y es esencial en lenguajes multi-plataforma como Haxe.

- Soporte nativo de **iteradores** para el recorrido directo de contenedores, posibilitando la implementación de iteradores personalizados.

Esta lista no es una lista exhaustiva de las características de Haxe pero sirve para que el lector se haga una idea de la potencia de este len-

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

guaje de alto nivel y de las facilidades que proporciona para el desarrollo de software[16].

Como todo buen lenguaje, Haxe tiene asociado una **biblioteca estándar** que comprende los tipos básicos del lenguaje, los contenedores, el soporte matemático y de expresiones regulares, la serialización o el manejo de flujos de E/S, entre otros muchos elementos[17].

Hola Mundo! con Haxe

Tradicionalmente, el primer programa que se discute cuando alguien se adentra en el estudio de un lenguaje de programación es el típico *Hola Mundo!*, cuya funcionalidad es la de simplemente imprimir un mensaje por la salida estándar (la pantalla).

El siguiente listado de código muestra este programa implementado con Haxe. Como se puede apreciar, la función *main* define el punto de entrada del programa en la línea 2, mientras que la función *trace* de la línea 3 se puede utilizar para mostrar la traza de un programa.

Listado 2.11: *Hola Mundo!* implementado con Haxe.

```
1 class Test {
2   static function main() {
3     trace("Hola Mundo!");
4   }
5 }
```

En este caso, *trace* se ha utilizado para mostrar el mensaje *Hola Mundo!* por la salida, la cual dependerá del *target* o lenguaje elegido a la hora de compilar. Note cómo la función *main* está enmarcada en la clase *Test*, definida en la línea 1.

El siguiente paso para ejecutar nuestro primer programa consiste en definir cuál será el lenguaje cuyo código generará el compilador de Haxe. Si se elige, por ejemplo, C++, entonces Haxe generará el código necesario para, posteriormente, compilarlo y ejecutarlo con el compilador de C++ instalado de manera nativa en la máquina. Para ello, es necesario crear un archivo *compile.hxml* cuyo contenido sea el que se muestra en el siguiente listado.

[16] En http://haxe.org/doc/features se muestra un listado más exhaustivo de las características de Haxe, de su biblioteca estándar y de las herramientas y bibliotecas más relevantes asociadas a dicho lenguaje.

[17] Vea http://haxe.org/api para una descripción más en profundidad de la API de Haxe.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Listado 2.12: Archivo con opciones de compilación de Haxe.

```
1 -cpp cpp
2 -debug
3 -main Test
```

Básicamente, Haxe hace uso del anterior archivo para conocer cuál será el lenguaje de programación de destino (línea 1), si se activará el modo de depuración (línea 2) y cuál será la clase en la que se encuentra el punto de entrada o función principal (línea 3).

Para compilar y generar un ejecutable para C++, tan sólo es necesario ejecutar el siguiente comando desde la línea de órdenes:

```
$ haxe compile.hxml
```

La figura 2.18 muestrar la jerarquía de archivos y directorios generados después de la ejecución del comando anterior.

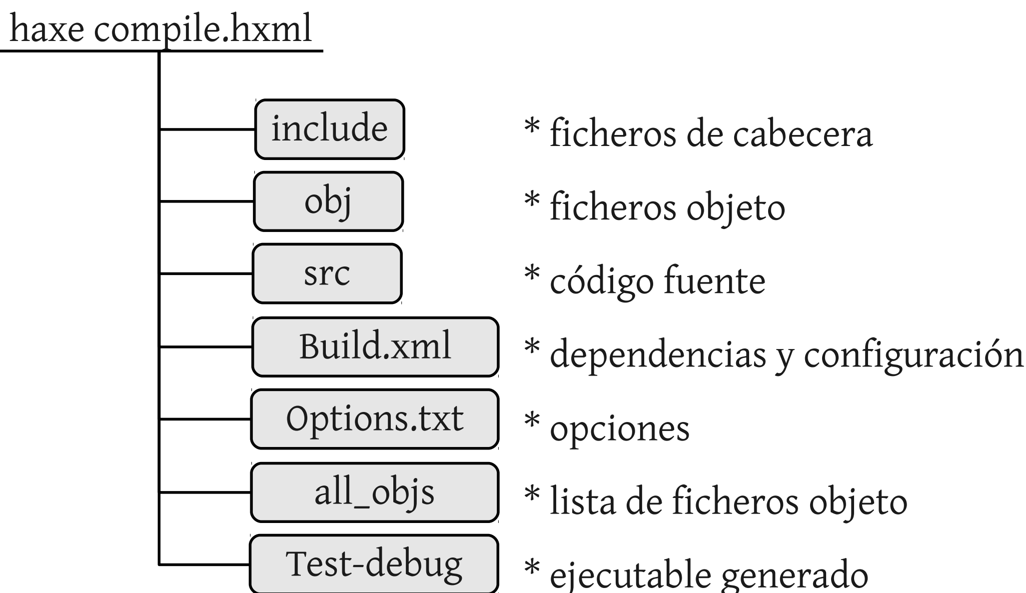

Figura 2.18: Estructura de archivos y directorios generados tras compilar el básico *Hola Mundo!*.

Si todo ha ido bien, este ejemmplo se puede probar mediante el siguiente comando:

```
$ ./cpp/Test-debug
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

La salida de este programa debería ser

```
Test.hx:3: Hola Mundo!
```

Note cómo la función *trace* tiene como objetivo principal servir como un mecanismo de traza o *log* ya que, además de mostrar el mensaje asociado, imprime el número de línea y la clase en la que se ejecutó.

2.3.3. Instalación y configuración de NME

Esta sección discute la instalación y configuración de NME considerando la generación de código para plataformas GNU/Linux (C++) y Android (Java). Para realizar estos procesos se ha elegido el sistema operativo Ubuntu 12.04 LTS (Long Term Support) con una arquitectura de 32 bits.

El primer paso consiste en descargar el **script de instalación** de NME disponible en *http://www.nme.io/download/* para Linux[18]. Una vez descargado es necesario descomprimirlo mediante la siguiente instrucción:

```
$ tar xzf NME-3.5.2-Linux.tar.gz
```

Al descomprimir el anterior archivo, ya es posible ejecutar el script de instalación con permisos de superusuario para instalar en el sistema las herramientas de las cuales depende NME. Para ello, simplemente hay que ejecutar la siguiente instrucción:

```
$ sudo ./nme-installer.sh
```

El proceso de instalación de NME está altamente automatizado y estructurado. En primero lugar, dicho script solicita la instalación de Haxe y Neko[19] Posteriormente, el script descarga e instala versión correspondiente de NME. El último paso consiste en la instalación de bibliotecas adicionales de uso común, como *actuate*, *swf* y *svg*.

A continuación, ya es posible realizar la **configuración de NME** mediante el comando *nme setup* y en función del *target* para el cual se generará código fuente. Por ejemplo, es posible llevar a cabo la configuración para **sistemas GNU/Linux** a través del siguiente comando:

```
$ sudo nme setup linux
```

[18] En el momento de la instalación, la versión de NME era 3.5.2
[19] Las versiones instaladas de Haxe y Neko fueron la 2.10 y la 1.82, respectivamente.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Figura 2.19: Captura de pantalla de un juego tipo *Whack-A-Mole* desarrollado con NME por Joshua Granick. El objetivo del juego es acumular puntos al realizar *click* con el ratón sobre los castores cuando éstos salen de los agujeros.

En el caso particular de Ubuntu, dicha instrucción delegará en el gestor de paquetes *apt* la instalación de todas las herramientas y bibliotecas necesarias para generar código fuente en C++ para sistemas GNU/Linux.

Para comprobar si la instalación y configuración de NME para sistemas GNU/Linux ha sido adecuada, el lector puede descargar un sencillo juego, titulado *Whack A Mole!*[20], y generar el ejecutable a partir de la generación previa de código en C++. Para ello, simplemente hay que descomprimir el código fuente, implementado en Haxe, y ejecutar la siguiente instrucción:

```
$ nme test "Whack A Mole.nmml" linux
```

Si todo ha funcionado correctamente, el resultado de la ejecución del juego debería ser similar al de la figura 2.19.

En el presente curso, además de utilizar GNU/Linux como *target* para generar los ejecutables asociados a los distintos prototipos de juegos, también se generarán paquetes que puedan ejecutarse sobre el sistema operativo **Android**. En este contexto, la generación de código para otra plataforma hace necesaria configuración de NME para dicha plataforma.

Antes de hacer uso de la opción *setup* del comando *nme* es necesario instalar el SDK de Java. En este curso se hará uso de Java 7 y de las facilidades que proporciona Ubuntu para llevar a cabo su instalación:

[20] http://www.joshuagranick.com/blog/2011/07/01/game-tutorial-whack-a-mole/

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java7-installer
```

A continuación, ya es posible proceder a la **configuración de NME para Android** haciendo uso del comando *nme*:

```
$ sudo nme setup android
```

El proceso de configuración de NME para Android es algo más tedioso ya que involucra la instalación del SDK, el NDK y Apache Ant. La instalación del SDK lanza la aplicación gráfica *Android SDK Manager* (ver figura 2.20) para que el usuario pueda elegir qué herramientas y versiones del API de Android instalar. En el caso particular del proceso de configuración descrito en esta sección se optó por la versión 2.2 de Android.

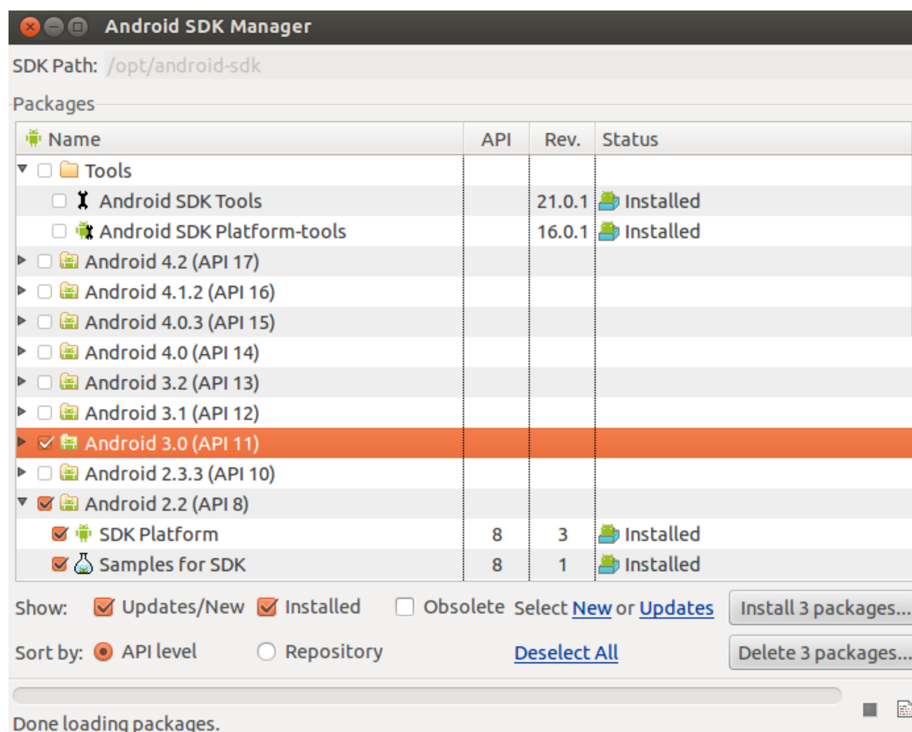

Figura 2.20: Captura de pantalla del *Android SDK Manager*. Esta herramienta permite administrar cómodamente qué herramientas y versiones del API de Android instalar en el sistema.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

El NDK de Android es un conjunto de herramientas que posibilita la implementación de partes de un programa haciendo uso de C y C++. Esta práctica puede resultar útil en términos de reutilización de código y/o mejora de la eficiencia.

La instalación de Apache Ant se puede realizar mediante el siguiente comando y de manera independiente a la configuración de NME para Android:

```
$ sudo apt-get install ant
```

Además de llevar a cabo la instalación de todas estas herramientas, el comando *nme setup android* también permite establecer, como paso final, la ruta a los directorios que contienen todas las herramientas y bibliotecas instaladas, junto con la ruta a las utilidades de Java 7 y Ant[21].

Finalmente, e igual que se discutió anteriormente para el caso de sistemas GNU/Linux, es posible comprobar si la instalación y configuración de NME para Android se completó de manera satisfactoria generando el paquete que contiene el ejecutable del juego *Whack A Mole!*. Para ello, simplemente es necesario ejecutar el siguiente comando:

```
$ nme test "Whack A Mole.nmml" android
```

Si todo ha funcionado con éxito, entonces se habrá generado un archivo denominado *WhackAMole-debug.apk* en el directorio *bin/android/bin/bin*, el cual se puede transferir directamente a un *smartphone* que haga uso de Android para poder ejecutarlo en el propio dispositivo[22].

Por otra parte, también es posible hacer uso del **emulador de Android** que incorpora el propio SDK previamente instalado y que se encuentra en el directorio *tools*. Sin embargo, antes es necesario crear un perfil o AVD (Android Virtual Device) que represente la configuración del dispositivo que se pretende emular (ver figura 2.21).

[21] En el caso de Ubuntu 12.04 LTS, los directorios del SDK y NDK de Android se instalaron en /opt. La ruta a las utilidades de Java 7 y Ant se estableció a /usr.

[22] Note que previamente es necesario activar la opción *Orígenes Desconocidos* en el menú de *Aplicaciones* del teléfono

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

Figura 2.21: Captura de pantalla del emulador de Android includo en el SDK.

Este perfil o AVD se puede crear de manera sencilla desde el menú *Tools->Manage AVDs...* del propio *Android SDK Manager*.

2.4. *Hello World!* con NME

Esta sección discute un **primer ejemplo** desarrollado con NME que sirva como punto de partida del lector para afrontar el tutorial de desarrollo del capítulo 3.

Básicamente, este primer *Hello World!* con NME integra dos aspectos esenciales en el desarrollo de videojuegos: i) la **animación básica** y ii) la **integración de sonido**. En el primer caso de este sencillo programa que se discutirá a continuación, la animación está asociada al movimiento de una imagen a lo largo del tiempo sobre la pantalla (ver figura 2.22). En el segundo caso, la integración de sonido está representada por la reproducción de un archivo de audio.

En ambos casos se ha hecho uso extensivo de las facilidades que proporciona NME para el desarrollo de videojuegos o, desde un punto de vista general, el desarrollo de aplicaciones gráficas interactivas.

El siguiente listado de código muestra la cabecera de la clase *HolaMundoNME*, la cual se instanciará posteriormente, sus variables de clase y el constructor.

En primer lugar, note cómo se define una **clase *HolaMundoNME*** que hereda de la clase *Sprite*, la cual forma parte del conjunto de clases que conforman la implementación de NME. Esta relación de herencia se ha establecido para facilitar el *rendering* de distintos elementos gráficos, como el logo asociado.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

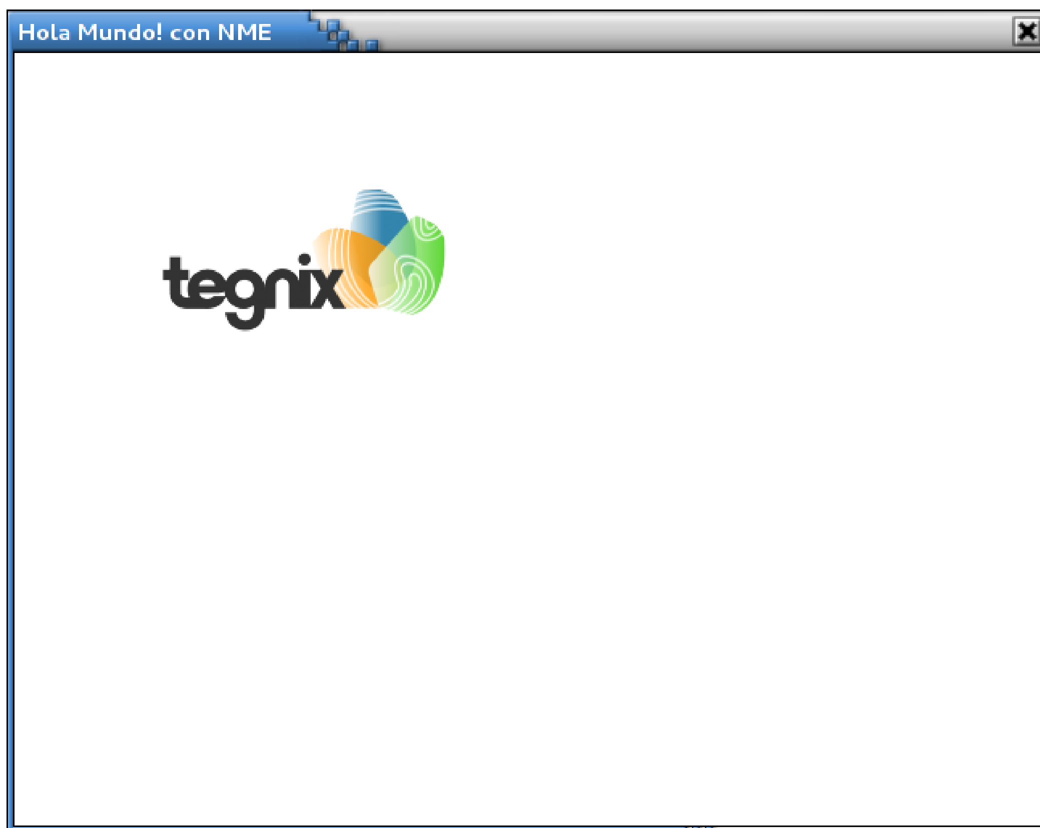

Figura 2.22: Resultado visual obtenido al ejecutar el *Hola Mundo!* con NME discutido en esta sección.

La clase *Sprite* de NME representa un bloque básico de construcción de listas de representación, es decir, representa un nodo que puede mostrar recursos gráficos y que puede albergar hijos.

A continuación se definen las **variables de clase** *logo* (línea 2) y *sound* (línea 3), las cuales son de tipo *Bitmap* y *Sound*, respectivamente. La primera de ellas se utilizará para cargar el logo, mientras que la segunda se usará para cargar la información asociada a un *track* de audio.

Por último, el **constructor** (función *new()* de la línea 6) se encarga de llamar al constructor de *Sprite* mediante *super()* (línea 8) y define el manejador de entrada mediante la función *this_onAddedToStage* (línea 10). Este manejador está asociado a un evento de NME de la categoría

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Event.ADDED_TO_STAGE, es decir, el código de la función denominada *this_onAddedToStage* se ejecutará cuando se produzca un evento de tipo *Event.ADDED_TO_STAGE*.

Listado 2.13: *Hola Mundo!* con NME. Constructor.

```
1  class HolaMundoNME extends Sprite {
2
3    private var logo:Bitmap;
4    private var sound:Sound;
5
6    public function new () {
7      /* Delega en el constructor de Sprite (clase padre) */
8      super ();
9      /* Define el manejador de entrada */
10     addEventListener (Event.ADDED_TO_STAGE, this_onAddedToStage);
11   }
```

Note cómo *addEventListener*, en la línea *10*, realiza la asociación entre dicho evento y el manejador, el cual representa el *event listener* asociado al mismo.

El concepto de *event listener* es muy común en el desarrollo de videojuegos y representa la relación entre la ocurrencia de un evento, por ejemplo el redimensionado de una ventana, y el código o manejador que se ejecutará cuando el primero tenga lugar. Este planteamiento es muy escalable y posibilita la delegación de eventos.

Precisamente, el siguiente listado de código muestra la implementación del manejador *this_onAddedToStage()* que, a su vez, delega la lógica en la función *construct()*. En el desarrollo de videojuegos, es común independizar la parte de **inicialización** y finalización de un videojuego en funciones explícitamente definidas para ello. Por este motivo se plantea la función *construct()*.

Listado 2.14: *Hola Mundo!* con NME. Función *onAddedToStage()*.

```
1  private function this_onAddedToStage (event:Event):Void {
2    construct ();
3  }
```

La función *construct()*, como muestra el siguiente listado, es la responsable de realizar la **carga efectiva del logo** en la línea 6 y de centrarlo en la pantalla con un nivel de transparencia inicial de 0 (líneas

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

9–11). Para ello, solamente es necesario acceder a las variables *x* e *y* de logo, el cual es de tipo *Bitmap*.

Listado 2.15: *Hola Mundo!* con NME. Función *construct()*.

```
1  private function construct ():Void {
2    stage.align = StageAlign.TOP_LEFT;
3    stage.scaleMode = StageScaleMode.NO_SCALE;
4
5    /* Carga del logo */
6    logo = new Bitmap (Assets.getBitmapData ("assets/tegnix.png"));
7
8    /* Centra el logo en el centro de la pantalla con transparencia a 0
         */
9    logo.x = (stage.stageWidth - logo.width) / 2;
10   logo.y = (stage.stageHeight - logo.height) / 2;
11   logo.alpha = 0;
12
13   /* Incorpora el logo a la display list */
14   addChild (logo);
15
16   /* Obtiene el sonido y lo reproduce*/
17   sound = Assets.getSound ("assets/music/elvuelodelmoscardon.ogg");
18   sound.play ();
19
20   /* Delega la animacion en la biblioteca Actuate */
21   Actuate.tween (logo, 2, { alpha: 1 } ).onComplete (moveLogo);
22 }
```

A continuación, en la línea 14 se añade el logo a la lista de representación que maneja internamente la clase *Sprite*, de la cual hereda *HolaMundoNME*.

Finalmente, es necesario asociar el **archivo de audio** que se reproducirá al ejecutar la aplicación y realizar la animación del logo. El primer problema se resuelve fácilmente gracias a la función *getSound()* de la clase *Assets* y a la función *play()* de la clase *Sound*. El segundo problema, es decir, la animación del logo, se delega en la función *tween* de la clase *Actuate*.

La **animación** juega con el nivel de transparencia del logo para que su aparición y desaparición sea suave, en lugar de que aparezca y desaparezca súbitamente. Para ello, los argumentos especificados en la línea 21 permiten que, por ejemplo, el logo pase de un estado de transparencia total a totalmente visible en 2 segundos.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

La biblioteca *Actuate* permite manejar de manera cómo qué debe ocurrirle a un objeto entre el intervalo de tiempo que transcurre desde el estado actual hasta un estado futuro deseado por el desarrollador.

Para mover el logo por la pantalla se utiliza la función *moveLogo*. Note cómo está función se ejecuta cada vez que se completa la función *tween*. Esta relación se establece mediante la función *onComplete*.

Listado 2.16: *Hola Mundo!* con NME. Función *moveLogo()*.

```
1 private function moveLogo ():Void {
2     var randomX = Math.random () * (stage.stageWidth - logo.width);
3     var randomY = Math.random () * (stage.stageHeight - logo.height);
4     Actuate.tween (logo, 2, { x: randomX, y: randomY } )
5     .ease (Elastic.easeOut)
6     .onComplete (moveLogo);
7 }
```

Básicamente, el movimiento del logo se realiza de manera aleatoria alterando las coordenadas x e y (líneas 2-3) de la posición de éste. De nuevo, se delega en *Actuate* la funcionalidad de realizar el movimiento efectivo mediante una ecuación cuadrática que aproxima la *parada* del logo (línea 5).

El **punto de entrada** de este sencillo *Hola Mundo!* está representado por la función *main* que, simplemente, crea una instancia de la clase definida.

Listado 2.17: *Hola Mundo!* con NME. Función *main()*.

```
1 public static function main () {
2   Lib.current.addChild (new HolaMundoNME ());
3 }
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es