Pruebas unitarias con CPPUnit

Veamos esta técnica propia de la eXtreme Programming, y cómo podemos utilizar CPPUnit para mejorar la calidad de nuestros proyectos en C++Builder.

Índice

Introducción

Las pruebas unitarias es uno de los métodos con el que puedes mejorar la calidad de tus sistemas de software.

En un sistema de calidad de software (o *Quality Assurance*), existen principalmente dos tipos de pruebas:

- **Pruebas unitarias (o de aceptación)**: comprobaciones que hacemos a las *unidades lógicas* de nuestro programa. Se verifica que una unidad funciona correctamente por sí misma, sin tener en cuenta las relaciones que pueda tener con otras partes del sistema.

- **Pruebas funcionales (o pruebas de sistema o integración)**: comprobamos el sistema globalmente, haciendo énfasis en las colaboraciones entre unidades. Se prueba cada una de las opciones (o casos de uso) que ofrece el sistema, pudiendo ser procesos automáticos, acciones sobre el interfaz gráfico, etc.

Vamos a hablar sobre las pruebas unitarias, y cómo podemos aplicarlas a nuestros proyectos en C/C++, a través del marco de pruebas CPPUnit.

En la definición que acabamos de dar, hemos hablado de *unidades lógicas*, aunque este concepto puede ser un poco ambiguo. Para ir entendiéndolo, diremos que las *unidades lógicas* de un programa son aquellas partes en que lo hemos dividido para entenderlo mejor. Pueden ser los módulos, paquetes, clases, subsistemas, funciones o cualquier otro mecanismo que nos ofrezca el lenguaje de programación que estamos utilizando. Nosotros, para simplificar, utilizaremos las clases como unidades lógicas.

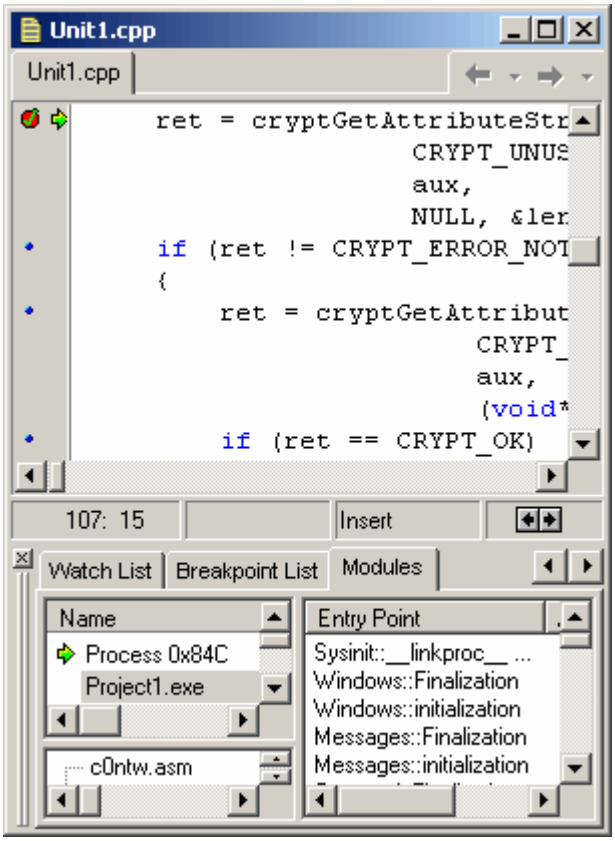
Las pruebas unitarias tienen un único requisito muy básico: que el programa que queremos probar tenga unidades lógicas. Aunque pueda parece obvio, pero no lo es tanto. He podido ver muchos, muchísimos programas en los que no existe ningún tipo de unidad lógica, especialmente en entornos RAD como Delphi, C++Builder, Visual Basic, etc. Lo único que existe es una ventana y mucho código en sus eventos, y a lo sumo alguna función de propósito general para estructurar el sistema. Como os podréis imaginar, en este escenario es imposible comprobar si un módulo cumple con su cometido, porque su funcionalidad está entremezclada, y no hay manera de aislar cada una de sus partes para verificar su funcionamiento.

Para aprender a separar un programa en unidades lógicas, es imprescindible aprender análisis y diseño, apoyándonos en técnicas y metodologías como UML, uso de patrones de diseño, Yourdon, Merisse, etc.

Pero... ¿en qué consisten las pruebas unitarias?

Es típico que los programadores verifiquemos un algoritmo través del depurador del entorno de desarrollo. Ejecutando paso a paso podemos ir comprobando el valor de la variables, y verificando así que cada variable toma el valor adecuado en cada momento. Este es un proceso lento y complicado, ya que requiere mucha atención del programador, para ir comprobando todas las variables, y es muy fácil perderse en algún detalle o equivocarse en alguna operación. Además, el proceso de depuración requiere mantener la atención al máximo durante mucho tiempo, y yo, no soy capaz de mantener ese nivel de concentración durante un proceso de depuración largo, de por ejemplo, una o dos horas.

Otra opción es desarrollar un pequeño programa que utilice la unidad que estamos probando. Una ventana con un botón, o un programa de consola, desde el que vamos codificando "lo que se nos ocurre", para ir viendo que todo funciona correctamente: llamadas a funciones o métodos, pruebas con casos "raros", usos típicos... una vez que hemos visto que funciona, lo típico es desechar el programa.



El último enfoque, y el más acertado, es el diseño y programación de pruebas unitarias, a través de "Casos de pruebas" y "Colecciones de pruebas".
La pruebas unitarias son pequeños módulos auxiliares, que se encargan de verificar el funcionamiento de otras unidades lógicas del sistema. Una "Colección de Pruebas" (*Test Suite*) es un conjunto de pruebas que hacemos sobre una unidad lógica. Si, por ejemplo, tenemos una unidad que hace operaciones matemáticas, deberíamos hacer sobre ella varias pruebas: una que compruebe que sume bien, otra que divida bien, otra que verifique que es capaz de manejar números negativos o decimales, etc. Cada una de estas pruebas individuales la llamaremos un "Caso de Prueba" (*Test Case*).

La forma de programar las pruebas es sencilla: simplemente basta con hacer una serie de llamadas a las funciones que queremos verificar, utilizando para ello valores conocidos en sus parámetros. Después comprobaremos que el valor retornado por la función es el que esperábamos, y si la función ha hecho algún cambio global (crear ficheros, cambios en base de datos, etc.), comprobaremos que estos cambios son los esperados. Más adelante veremos un ejemplo concreto.

Las colecciones de pruebas se debe mantener junto con el código fuente, para ir ampliándola según se amplía la unidad. Por ejemplo, si aparece un nuevo método público, tendremos que modificar la colección de pruebas, para hacer un

nuevo caso de prueba. Cada vez que aparezca un nuevo bug, debemos realizar un nuevo caso de prueba que verifique que el bug no ha vuelto a aparecer, simulando todos los pasos necesarios para reproducirlo.

Los gurús de la Programación eXtrema (www.extremeprogramming.org), y en general de las metodologías de desarrollo guiado por pruebas (TDD: *Test Driven Development*), recomiendan desarrollar las pruebas unitarias antes que la propia unidad. La principal razón de esto es muy sencilla: durante la programación, pasamos por muchas fases de investigación y descubrimiento de nuestras propias necesidades. Muy pocas veces desarrollamos algo con el conocimiento absoluto de lo que queremos conseguir y a dónde queremos llegar, sino que este conocimiento se va adquiriendo mientras escribimos el código. Si programamos las pruebas antes, vamos definiendo cómo queremos utilizar la unidad, qué queremos pasarle como parámetros de entrada y qué nos gustaría que nos respondiese como salida, en definitiva: el conocimiento que se obtiene durante la programación. Lo más habitual es ir adquiriendo este conocimiento durante la vida del proyecto, mejorándolo poco a poco en cada una de las iteraciones (o versiones). Sin embargo, si lo primero que hacemos es programar las pruebas, conseguimos este conocimiento antes de desarrollar el producto, por lo que una vez que hemos terminado las pruebas, ya sabemos lo suficiente como para programar la unidad de una forma mucho más rápida y consistente que de la otra forma.

Ventajas de las pruebas unitarias

La programación de pruebas unitarias puede ser una tarea laboriosa, ya que requiere un tiempo que no tendríamos que invertir en caso de no hacerlas.

Desarrollar software sin pruebas unitarias es como jugar al fútbol en la nieve: conforme más pasa el tiempo, más grande se hace la bola y más difícil es manejarla, hasta que llega un momento en que la pelota es más grande que tú y ya no puedes seguir. Sin embargo, utilizando pruebas unitarias es como cuando juegas al fútbol en la playa, al principio te cuesta más, pero cuando tus piernas se han acostumbrado a la arena, puedes seguir jugando sin que la pelota vaya haciéndose más grande.

Aunque al principio te cueste tiempo acostumbrarte a este método, las ventajas de usar este tipo de pruebas son muchas, entre ellas podemos decir:

- **Los errores son más fáciles de localizar**: bastará con ejecutar la batería de "Colecciones de pruebas", y ver qué módulos no las pasan.
- **Los errores están más acotados**: cuando un programa falla, muchas veces no sabemos por donde pueden venir los problemas. Con las pruebas unitarias conseguimos acotar los errores, sabiendo qué módulos no están pasando las pruebas unitarias.
- **Se reducen los "efectos secundarios"**: muchas veces, cuando queremos arreglar algo bajo presión, cometemos otros errores, o no tenemos en cuenta ciertos aspectos, que hacen que el programa deje de funcionar por otro sitio. Incluso a veces, es más peligroso arreglar un error que dejarlo como está, ya que podemos subsanar el error, pero generar otros distintos. Aplicando las pruebas unitarias, es más fácil controlar a "ese potro

salvaje" que tenemos por programa, ya que pasando de nuevo la batería de pruebas nos aseguraremos de que todo funciona tal y como esperábamos.

- **Se da más seguridad al programador**: normalmente, la persona que ha programado un módulo no es la misma que la que tiene que corregir sus errores. Esto crea una sensación de inseguridad al programador, ya que, a la hora de corregir un error, no tiene la certeza de que su corrección no va a afectar a otros módulos que desconoce. Las pruebas unitarias aseguran que una corrección no repercute en otros módulos, y permite al programador centrarse en la corrección del error, y no en la repercusión que puede tener esa corrección.

- **Los errores se detectan antes que de otra forma**: Cuanto más tiempo permanece un bug en el sistema, más tiempo requiere eliminarlo y más difícil se hace su resolución, ya que el impacto que puede causar la corrección es mucho mayor. De hecho, con pruebas unitarias, la mayoría de los errores de programación se detectan durante la propia etapa de programación, ya que esta no se da por concluida hasta que la unidad pasa su batería de pruebas unitarias.

- **Las pruebas funcionales se hacen más sencillas**, ya que la mayoría de los aspectos individuales de cada unidad ya están probados a través de las pruebas unitarias. De este modo, las pruebas funcionales deben centrarse sólo en verificar la correcta cooperación de las distintas unidades, y en los funcionamientos generales del programa.

- **El programador escribe código de una forma más lógica**: cuando un programador sabe que va a tener que escribir pruebas unitarias sobre su software, lo diseña de una forma mucho más simple y accesible para las pruebas, o en definitiva: escribe código más limpio y comprobable. Esto es debido a que no se crean más dependencias de las necesarias, porque esas dependencias dificultarían mucho las pruebas. Cada vez que parezca necesitarse una nueva dependencia (por ejemplo para llamar a un método de otra clase), el programador se lo va a pensar dos veces, y valorará si realmente es necesario o si hay otros caminos. En muchas ocasiones, se dará cuenta de que hay dependencias que se pueden evitar, consiguiendo así una arquitectura con módulos mucho menos acoplados (con menos dependencias).
  Además, si aplicamos la metodología de escribir las pruebas antes que el propio código, este efecto se multiplicará, ya que al escribir las pruebas estaremos haciendo el primer uso del código que vamos a desarrollar después, y como el interfaz público todavía no está definido, lo iremos definiendo conforme escribimos las pruebas.

- **Cada prueba se convierte en un ejemplo de uso**. Seamos sinceros: cuando vamos a utilizar una nueva librería o conjunto de clases, lo primero que solemos hacer es buscar los ejemplos en la documentación y ver si hay alguno que encaje en el uso que queremos darle. A veces lo hay (Microsoft suele poner mucho cuidado en esto), y otras veces los ejemplos son tan triviales que no sirven ni para hacer el "Hola mundo" (y Borland es experto en esto). Con el conjunto de pruebas, ponemos a disposición del programador-usuario un conjunto bastante amplio de ejemplos, que si están completos, abarcan todos los posibles usos de la clase. Además, tenemos la seguridad que siguiendo esos ejemplos, el código funcionará, ya

que hay pruebas que nos garantizan su funcionamiento correcto. En definitiva, con las pruebas unitarias matamos dos pájaros de un tipo: probamos nuestro código y escribimos ejemplos para que los demás los consulten.

Diseño de pruebas unitarias

Ya hemos dicho que los casos de prueba son unidades que se encargan de realizar las pruebas de otras unidades, simplemente utilizándolas y verificando que se comportan como deberían.

Para que esto sea más tangible, vamos a poner un ejemplo “casi real”: estamos en un sistema que tiene un módulo completo para cálculos aritméticos. Como estamos programando en C (todavía no sabemos qué es esa cosa del C++), tenemos en este módulo una unidad “suma” que se representa por una única función: sumar(a, b).

```
int sumar(int a, int b)
{
    return (a + b);
}
```

Podemos imaginar que pertenecemos al departamento de calidad (QA) de una gran empresa de desarrollo, y nos ha tocado desarrollar las pruebas unitarias para esta unidad que suma. Nuestra tarea consiste en escribir las pruebas, y lo haremos desde otra función, que retornará FALSE si alguna de las pruebas falla.

La función tiene un esquema fijo: una serie de sentencias de prueba en las que se verifica el caso correcto. En caso de retornarse un valor distinto al esperado, se retorna el valor de error.

```
int ProbarSumar()
{
    if ( sumar(1, 2) != 3 )
        return (FALSE);

    if ( sumar(0, 0) != 0 )
        return (FALSE);

    if ( sumar(10, 0) != 10 )
        return (FALSE);

    if ( sumar(-8, 0) != -8 )
        return (FALSE);

    if ( sumar(5, -5) != 0 )
        return (FALSE);

    if ( sumar(-5, 2) != -3 )
        return (FALSE);

    if ( sumar(-4, -1) != -5 )
        return (FALSE);

    return (TRUE);
}
```

Después de esto, no se trata más que de llamar a esta función desde el main, y mostrar un mensaje indicando el éxito o fracaso.

Como podéis ver, hemos probado los casos más típicos de suma:

- Positivo + Positivo
- Cero + Cero
- Positivo + Cero
- Negativo + Cero
- Positivo + Negativo
- Negativo + Positivo
- Negativo + Negativo

Cada una de las pruebas se compara con el resultado correcto, así, si la función "suma" retorna algún valor incorrecto, la función de prueba retornará un FALSE. En caso de pasarse todas las pruebas, la función retornará TRUE.

Esta prueba que acabo de hacer se llama "Caso de prueba" (*Test Case*) y representa a un conjunto de verificaciones que hacemos sobre la misma unidad. Todas estas verificaciones deben estar relacionadas con un mismo aspecto (o caso de uso) de la unidad a probar, en nuestro caso: verificar los resultados de la suma teniendo en cuenta el signo de los operandos.

Podemos hacer distintos casos de prueba sobre la misma unidad, por ejemplo, cuando queremos probar distintos aspectos. En nuestro ejemplo podemos crear otro caso de prueba para probar las propiedades de la suma:

```
int ProbarPropiedadesSumar()
{
     // conmutativa: a + b = b + a
     if ( sumar(1, 2) != sumar(2, 1) )
         return (FALSE);

     // asociativa: a + (b + c) = (a + b) + c
     if ( sumar(1, sumar(2, 3)) != sumar(sumar(1, 2), 3) )
         return (FALSE);

     // elemento neutro: a + NEUTRO = a
     if ( sumar(10, 0) != 10 )
         return (FALSE);

     // elemento inverso: a + INVERSO = NEUTRO
     if ( sumar(10, -10) != 0 )
         return (FALSE);

     return (TRUE);
}
```

En este caso de prueba hemos verificado que se cumplen las propiedades típicas de la suma. Lo siguiente que tendríamos que hacer es añadir este nuevo caso de prueba a la misma función *main* de antes, y mostrar el resultado de ambas funciones.

Así, nuestro programa de pruebas será algo así como esto:

```
int main(int argc, char** argv)
{
    int ok, err;
    printf("Casos de prueba sobre la unidad suma(a, b):");

    ok = 0;
    err = 0;

    printf("\r\n\tProbando signos de la suma... ");
    if ( ProbarSumarSignos() ) {
        printf("ok!");
        ok++;
    }

    else {
        printf("ERROR");
        err++;
    }

    printf("\r\n\tProbando propiedades de la suma... ");
    if ( ProbarPropiedadesSumar() ) {
        printf("ok!");
        ok++;
    }
    else {
        printf("ERROR");
        err++;
    }

    /* añadir otros casos de pruebas */

    printf("\r\n\r\nResumen de las pruebas sobre la unidad suma:");
    printf("\r\n\tCorrectas: %d", ok);
    printf("\r\n\tErroneas:  %d", err);

    return (err);
}
```

Como ya dijimos, al conjunto de casos de prueba sobre una misma unidad se le llama “Colección de Pruebas” (*Test Suite*).

Condiciones de error

El principal objetivo de las pruebas es averiguar condiciones bajo las que una unidad falla estrepitosamente. Esto, dicho de palabra, puede ser sencillo, pero transformar esas condiciones en código no siempre es fácil. Para comprobar el éxito o fracaso de una unidad, podemos utilizar tres tipos de comprobación:

- **Comprobar el retorno**: lo más sencillo para averiguar si una operación ha funcionado o no es comprobar su retorno. La mayoría de las funciones retornan un valor para indicar que la ejecución ha sido correcta o que ha ocurrido algún tipo de error. En nuestro ejemplo de la suma, hemos utilizado este tipo de comprobación.

- **Comprobar el estado**: una vez que ha terminado la función, normalmente se ha establecido alguna variable para indicar que ha cambiado el estado. Por ejemplo, las llamadas a los métodos de una clase suelen modificar atributos privados de la clase. Una buena manera de comprobar que todo ha ido bien, es comprobar que los atributos tienen los valores correctos después de la llamada a un método. Esto a veces no es posible, ya que los atributos internos de una clase suelen ser privados, pero eso es otra batalla.

- **Comprobaciones externas**: algunas unidades lógicas dependen y hacen modificaciones sobre otras unidades, sobre ficheros de texto, bases de datos, etc. En ese caso, es bueno comprobar el estado de estos elementos, por ejemplo comprobando que las tablas de base de datos tienen los valores correctos, o que los ficheros generados son los esperados.

Desarrollo guiado por pruebas

Anteriormente hemos dicho que el eXtreme Programming (XP) propone un modelo de pruebas ligeramente distinto. Bien, expliquemos esto con detalle porque merece la pena.

Una de las metodologías ágiles existentes es la llamada *Test Driven Development*, (o TTD) es decir: desarrollo guiado por pruebas. Esta metodología, (una de las que inspiró a Kent Beck a la hora de concebir el XP) propone realizar las pruebas unitarias antes que la propia unidad.
Espera un momento... ¿cómo vamos a probar algo que todavía no existe? Es como si intentamos medir la velocidad máxima de un coche antes de que se haya fabricado el primer ejemplar. Pues aunque suene raro, y no sea posible realizar en otros ámbitos, en el mundo del desarrollo de software esto es viable, y además muy recomendable.

Esta metodología está muy pautada y hay que seguir los siguientes pasos:

1. Pensar en la unidad o funcionalidad que queremos desarrollar, centrándonos en cómo nos gustaría que se usase desde el exterior. Una buena manera de hacer esto es imaginarnos que vamos a comprar esa unidad a una empresa y podemos definir cómo queremos usarla. El cliente siempre manda ¿no? pues imaginemos que somos los clientes y van a programarnos esa unidad a nuestro gusto.

2. Escribir el pseudocódigo de uno o varios ejemplos de su uso más habitual. Cada uno de estos ejemplos se llama en análisis "Caso de Uso", ya que define uno de los casos en que se usará la unidad que estamos analizando. No detallaremos ni escribiremos ejemplos de usos extraños, sino los ejemplos típicos de uso. Como vamos a ser los que usemos la futura unidad, intentaremos que el uso sea lo más sencillo posible a la vez que flexible y potente.
   Cuando terminemos este paso, tendremos una lista de Casos de Uso, además de una lista de tareas a completar. De este modo, como dicen en mi pueblo, matamos dos pájaros de un tiro: hemos obtenido el análisis de casos de uso (incluso podríamos representarlo con un diagrama UML de Casos de Uso), y hemos confeccionado una lista de tareas a completar para dar por finalizada la unidad.

3. Codificar el pseudocódigo de cada uno de los ejemplos en forma de Caso de Prueba, verificando en todo momento los retornos de los métodos que hemos definido en el paso anterior. De este modo, cada Caso de Uso tiene su correspondiente Caso de Prueba que lo verifica. Durante esta codificación refinaremos el uso que hemos imaginado, añadiendo o quitando parámetros, modificando los retornos de los métodos, etc.

4. Compilar ese Caso de Prueba. Lógicamente, fallará la compilación, ya que se están haciendo llamadas a métodos o unidades que no existen.

5. Codificar todas aquellas clases/funciones/métodos/lo-que-sea para que la prueba compile. Es importante que todos los métodos que

codifiquemos retornen un valor de error, que dependiendo del método en cuestión puede ser `false`, -1, o cualquier otra cosa.

6. Compilar otra vez el Caso de Prueba: ahora tiene que compilar correctamente ya que en el paso anterior hemos añadido todo lo necesario.

7. Ejecutar la prueba: fallará estrepitosamente, porque todos los métodos o funciones están vacíos y retornan un valor de error. Si alguna prueba pasa correctamente, significará que la prueba está mal escrita.

8. Codificar cada una de las funciones/métodos para que todas las pruebas vayan pasando. Posiblemente no pasarán a la primera, sino que las pruebas nos irán indicando si el código que escribimos va por buen camino o no.

9. Daremos por finalizado el ciclo cuando todas las pruebas hayan pasado. En ese momento estaremos seguros de que nuestra unidad funciona para los Casos de Uso que hemos probado.

10. Opcionalmente podemos añadir nuevas pruebas para verificar más a fondo la unidad: añadir nuevos casos de uso más improbables (aunque posibles), pasar valores extremos a los métodos (negativos cuando se esperan positivos, cadenas vacías, punteros nulos, etc.), comprobando que la situación se controla y se retorna el error correspondiente, etc.

Si nunca habéis aplicado esta metodología, quizá estos pasos os resulten demasiado extraños, o no quede muy claro cómo habría que aplicarlo a un caso real. Pongamos un ejemplo. Supongamos que nos han encargado realizar una unidad que se encargue de enviar correos electrónicos. Esa unidad irá integrada en un sistema más grande, concretamente en la parte que se encarga de enviar informes de error o sugerencias de los usuarios que están usando el programa. Aplicando la metodología TTD seguiremos los siguientes pasos:

1. Pensar en la funcionalidad que queremos desarrollar: si lo que tenemos que desarrollar es un *algo* que envíe correos electrónicos, uno de los Casos de Uso será *enviar un correo electrónico*. Además nos han dicho que se tiene que permitir el envío de hasta un archivo adjunto, así que otro caso de uso será *enviar un correo con un archivo adjunto*.

2. Escribir el pseudocódigo de uno o varios casos de uso.
   Parar enviar un correo electrónico:

```
// Para enviar un correo electrónico es necesario conocer
// un servidor SMTP.
// Con esta función establecemos los datos del servidor
// SMTP a través del que enviaremos el correo
EstablecerServidorEnvio(ip, puerto)

// Con esta llamada se enviará el correo
EnviarCorreo(dirección origen, dirección destino, asunto, cuerpo)
```

Para enviar un correo con adjunto:

```
// Establecemos el servidor de envío
EstablecerServidorEnvio(ip, puerto)

// Cargamos el adjunto y los metemos en un buffer
adjunto = CargarAdjunto("C:\ruta\nombre.ext");

// Tenemos que añadir un nuevo parámetro para pasar el adjunto
// Si no queremos enviar adjuntos pasaremos NULL
EnviarCorreo(origen, destino, asunto, cuerpo, adjunto);
```

Como veis, nos han salido varias funciones/métodos que tenemos que implementar, obteniendo así una lista de tareas. Entre las tareas a completar está la implementación de las funciones: `EstablecerServidorEnvio`, `CargarAdjunto` y `EnviarCorreo`.

3. Codificar ese pseudocódigo de cada uno de los ejemplos en forma de Caso de Prueba.
   Parar enviar un correo electrónico:

```
int ProbarEnviarCorreo()
{
    char *ip;
    int  puerto;
    char *origen, *destino, *asunto, *cuerpo;

    // datos del servidor
    ip = "127.0.0.1";
    puerto = 23;

    // si la función retorna error, no pasamos la prueba
    if ( !EstablecerServidorEnvio(ip, puerto) )
        return (FALSE);

    // datos del correo
    origen = "yo@mismo.com";
    destino = "tu@mismo.com";
    asunto = "un correo de prueba";
    cuerpo = "texto del correo de prueba\r\nChao pescao.";

    // el último parámetro indica que no se envía adjunto
    if ( !EnviarCorreo(origen, destino, asunto, cuerpo, NULL) )
        return (FALSE);

    return (TRUE);
}
```

Para enviar un correo electrónico con adjuntos:

```
int ProbarEnviarCorreoConAdjunto()
{
    char  *ip;
    int   puerto;
    char  *origen, *destino, *asunto, *cuerpo;
    char  *ruta;
    void  *adjunto;

    // datos del servidor
    ip = "127.0.0.1";
    puerto = 23;

    if ( !EstablecerServidorEnvio(ip, puerto) )
        return (FALSE);

    // datos del adjunto
    ruta = "C:\\fichero.ext";
    adjunto = CargarAdjunto(ruta);
    if (adjunto == NULL)
        return (FALSE);

    // datos del correo
    origen = "yo@mismo.com";
    destino = "tu@mismo.com";
    asunto = "un correo de prueba";
    cuerpo = "texto del correo con adjunto\r\nChao pescao.";

    if ( !EnviarCorreo(origen, destino, asunto, cuerpo, adjunto) )
        return (FALSE);

    return (TRUE);
}
```

4. Compilar ese Caso de Prueba. La compilación falla porque no encuentra las funciones `EstablecerServidorEnvio`, `CargarAdjunto` o `EnviarCorreo`.

5. Declararemos las funciones, dejando el cuerpo vacío y que retorne error en todas ellas.

6. Compilamos de nuevo las pruebas: ahora funcionan porque ya se encuentran las funciones.

7. Ejecutamos las pruebas: fallarán todas porque lo único que hacen las funciones es retornar error.

8. Codificaremos las tres funciones. Durante la codificación nos damos cuenta de algún detalle que no hemos tenido en cuenta: la función `CargarAdjunto` debe retornar el número de bytes que ocupa el archivo adjunto, además ese número de bytes hay que pasarlo también a la función `EnviarCorreo`. También necesitamos darle un nombre al fichero adjunto (porque el protocolo SMTP, para enviar correos, así lo requiere). Esto supone cambios en el prototipo de las funciones, así que tendremos que modificar también las pruebas. Finalmente, decidimos definir una estructura que represente el fichero adjunto, y pasaremos esta estructura a la función `EnviarCorreo`.

Finalmente, las prueba de envío de correo con adjunto quedará así:

```
int ProbarEnviarCorreoConAdjunto()
{
    char  *ip;
    int   puerto;
    char  *origen, *destino, *asunto, *cuerpo;
    char  *ruta;
    ADJUNTO adjunto;

    // datos del servidor
    ip = "127.0.0.1";
    puerto = 23;

    if ( !EstablecerServidorEnvio(ip, puerto) )
        return (FALSE);

    // datos del adjunto. Ahora se usa un struct
    adjunto.ruta = "C:\\fichero.ext";
    adjunto.datos = NULL; // se rellenará CargarAdjunto
    adjunto.size = 0;     // se rellenará CargarAdjunto

    if ( !CargarAdjunto(&adjunto) )
        return (FALSE);

    // datos del correo
    origen = "yo@mismo.com";
    destino = "tu@mismo.com";
    asunto = "un correo de prueba";
    cuerpo = "texto del correo con adjunto\r\nChao pescao.";

    // ahora se pasa la estructura del adjunto
    if ( !EnviarCorreo(origen, destino, asunto, cuerpo, &adjunto) )
        return (FALSE);

    return (TRUE);
}
```

9. Después de estas pequeñas correcciones, ejecutaremos las pruebas las veces que sean necesarias hasta que pasen correctamente, momento en el que daremos por terminado el ciclo.

10. Para asegurarnos bien, verificaremos cómo se comportan las funciones antes parámetros incorrectos:

```
int ProbarEnviarCorreo()
{
    char *ip
    int puerto;
    char *origen, *destino, *asunto, *cuerpo;

    // datos del servidor
    ip = "127.0.0.1";
    puerto = 23;

    // datos del correo
    origen = "yo@mismo.com";
    destino = "tu@mismo.com";
    asunto = "un correo de prueba";
    cuerpo = "texto del correo de prueba\r\nChao pescao.";

    // llamar a EnviarCorreo antes de establecer el servidor
    // Debe retornar error. Si retorna ok, no pasa la prueba
    if ( EnviarCorreo(NULL, destino, asunto, cuerpo, NULL) )
        return (FALSE);
```

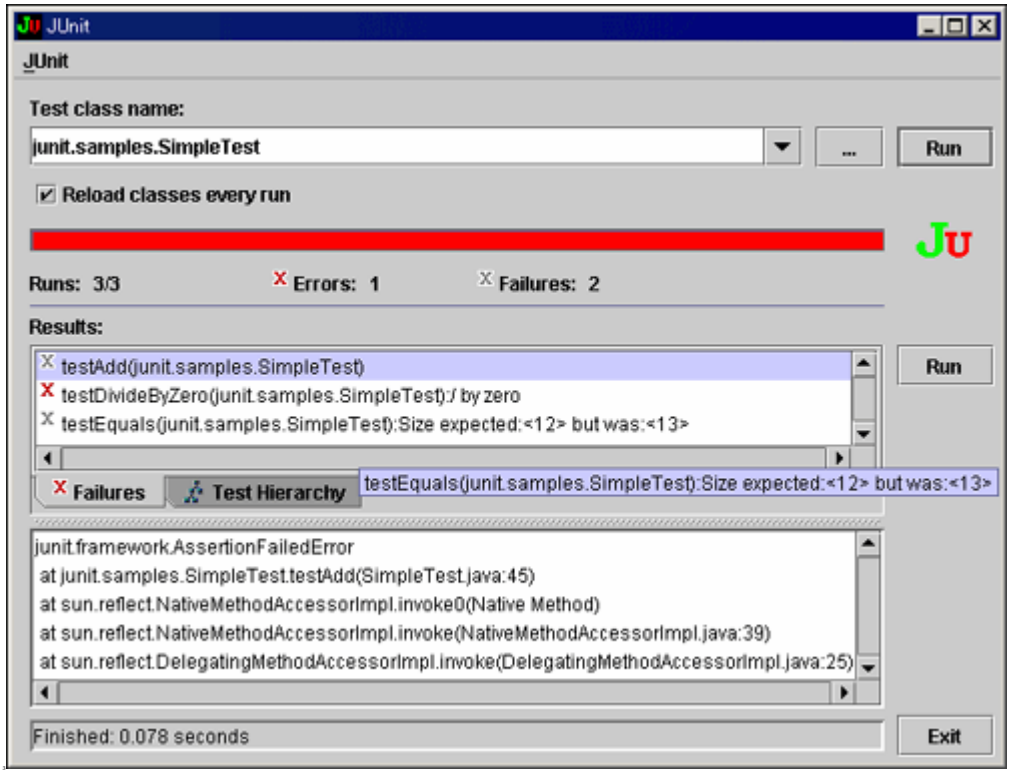
```
    // dirección ip incorrecta.
    // Si retorna Ok, no pasa la prueba.
    if ( EstablecerServidorEnvio(NULL, puerto) )
        return (FALSE);
    if ( EstablecerServidorEnvio("", puerto) )
        return (FALSE);

    // puerto incorrecto
    if ( EstablecerServidorEnvio(ip, -1) )
        return (FALSE);
    if ( EstablecerServidorEnvio(ip, 0) )
        return (FALSE);

    // caso correcto
    if ( !EstablecerServidorEnvio(ip, puerto) )
        return (FALSE);

    // datos del correo
    origen = "yo@mismo.com";
    destino = "tu@mismo.com";
    asunto = "un correo de prueba";
    cuerpo = "texto del correo de prueba\r\nChao pescao.";

    // dirección origen incorrecta
    if ( EnviarCorreo(NULL, destino, asunto, cuerpo, NULL) )
        return (FALSE);
    if ( EnviarCorreo("yo", destino, asunto, cuerpo, NULL) )
        return (FALSE);
    if ( EnviarCorreo("", destino, asunto, cuerpo, NULL) )
        return (FALSE);

    // dirección destino incorrecta
    if ( EnviarCorreo(origen, NULL, asunto, cuerpo, NULL) )
        return (FALSE);
    if ( EnviarCorreo(origen, "yo", asunto, cuerpo, NULL) )
        return (FALSE);
    if ( EnviarCorreo(origen, "", asunto, cuerpo, NULL) )
        return (FALSE);

    // asunto incorrecto
    if ( EnviarCorreo(origen, destino, NULL, cuerpo, NULL) )
        return (FALSE);

    // cuerpo incorrecto
    if ( EnviarCorreo(origen, destino, asunto, NULL, NULL) )
        return (FALSE);

    // caso correcto
    if ( !EnviarCorreo(origen, destino, asunto, cuerpo, NULL) )
        return (FALSE);

    return (TRUE);
}
```

Como habéis visto, durante las propia etapa de pruebas nos hemos dado cuenta de ciertos errores de diseño. Si no hubiéramos aplicado la metodología TTD, habríamos codificado la unidad para enviar correos la primera vez, y después nos habríamos dado cuenta de que no es correcta y tendríamos que haberla codificado por segunda vez. Escribiendo las pruebas antes, hemos analizado y diseñado más detalladamente la unidad, así que en el momento de codificarla tenemos una idea mucho más clara y precisa de lo que debe hacer y cómo lo debe hacer.

Herramientas de pruebas

Hasta ahora hemos estado desarrollando pruebas, aunque no hemos utilizado ninguna herramienta especial para ello. Simplemente hemos escrito una función C que retorna `FALSE` si la prueba no funciona correctamente. Esa función C la llamamos desde la función `main` que lo único que hace es mostrar en la pantalla el resultados de las distintas pruebas y llevar unas pequeñas estadísticas. Aunque la esencia es la misma, este no es el método más correcto para escribir las pruebas.

A lo largo de los años, los programadores más experimentados en la metodología TTD han desarrollado métodos y herramientas para escribir las pruebas más cómodamente. Dos de estos desarrolladores fueron Kent Beck y Eric Gamma (dos eminencias en el campo, uno por desarrollar la eXtreme Programming y el otro por su libro Design Patterns, que describe los patrones comúnmente llamados GoF), quienes desarrollaron una colección de clases para Java llamada JUnit (www.junit.org). Con estas clases, podemos desarrollar nuestros casos y colecciones de prueba fácilmente, heredando de sus propias clases base y utilizando los mecanismos que nos proporciona. Además, ofrece una serie de interfaces gráficos para visualizar estas pruebas, ejecutarlas, ver sus resultados, seleccionar aquellas que queremos ejecutar, etc.



A estas colecciones de clases, junto con sus herramientas se las conoce como "Testing Frameworks", o "Marcos de pruebas", ya que gracias a ellas, tenemos toda la infraestructura necesaria para desarrollar pruebas unitarias de forma rápida, cómoda, extensible y fiable.

JUnit ha tenido tanto éxito que se ha extendido a otros muchos lenguajes de programación, gracias al trabajo desinteresado de muchos programadores. Todos los frameworks heredados de JUnit han recibido la denominación xUnit, con la que se indica que se trata de una migración, y se siguen las normas que marcó JUnit. Entre los framworks xUnit, existen versiones para C/C++ (CUnit y CPPUnit), Delphi (DUnit), PHP (PHPUnit), HTML (HTMLUnit), NUnit (plataforma .NET), VBUnit (Visual Basic) y un largo etc.

El modo de trabajar de todos los frameworks xUnit es parecido entre ellos, aunque cada uno con las peculiaridades de su propio lenguaje. La idea principal ya la hemos explicado: se trata de desarrollar una unidad que se encargue de probar a otra unidad. Para programar esta prueba, se hace un uso intensivo de la unidad que queremos probar, verificando en todo momento que se comporta como esperábamos.

Para el mundo C/C++ existen varios frameworks, unos más conocidos que otros. El principal y más conocido es CPPUnit (http://cppunit.sourceforge.net), que trataremos a continuación, aunque también deberías considerar el uso de los siguientes:

- Boost.Test http://boost.org/libs/test/doc/index.html
- CPPUnitLite http://c2.com/cgi/wiki?CppUnitLite
- NanoCPPUnit: http://www.xpsd.org/cgi-bin/wiki?NanoCppUnit
- Unit++: http://unitpp.sourceforge.net/
- CxxTest: http://cxxtest.sourceforge.net/

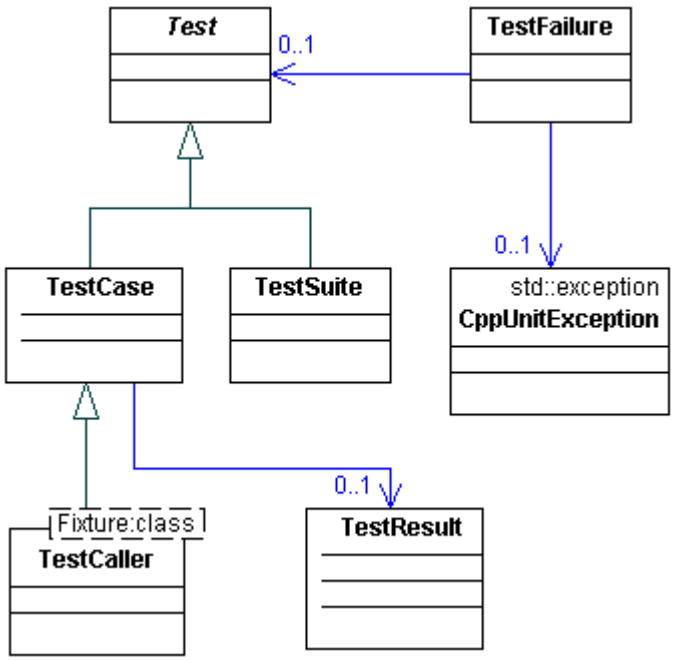
La página de Noel Llopis (un desarrollador de juegos en C++) tiene una interesante comparativa de *test frameworks* (marcos de pruebas):

http://www.gamesfromwithin.com/articles/0412/000061.html

El framework CPPUnit

CPPUnit utiliza orientación a objetos (como el lenguaje para el que se creó originalmente), así que trabajaremos con conceptos como herencia, encapsulación y polimorfismo. Además de orientación a objetos, también se utiliza continuamente la gestión estructurada de excepciones de C++, así que debemos saber utilizarlas correctamente, junto con las cláusulas try-finally y try-catch.

Cada caso de prueba se codifica en una clase derivada de la clase "TestCase". Esta clase nos proporciona los elementos básicos para programar el caso de prueba, registrarlos dentro de una colección de pruebas, ejecutarlo, etc.



Para ir viendo cómo utilizar CPPUnit, vamos a poner un pequeño ejemplo. Supongamos que estamos desarrollando una clase (para nosotros se trata de una unidad lógica) cuya principal responsabilidad es guardar y recuperar una serie de datos (agrupados en una estructura) en disco. Esta clase se llama "DatosDisco" y podéis verla en el siguiente listado:

```
typedef struct _DATOS
{
    int  numero;
    char cadena[256];
} DATOS, *LPDATOS;

class DatosDisco
{
public:
    DatosDisco();
    ~DatosDisco();

    LPDATOS getDato();
    void setDato(LPDATOS value);

    bool leer(char *archivo);
    bool guardar(char *archivo);
private:
    DATOS dato;
};
```

Preparación de CPPUnit

La versión que vamos a utilizar no es la oficial de CPPUnit, sino que es una adaptación especial para C++Builder desarrollada inicialmente por Marco Dorantes (www.angelfire.com/dc/marcodorantes/), con algunos cambios que he hecho yo mismo. Podéis encontrar esta versión en un enlace al final del artículo. Los conceptos básicos son los mismos que en la versión oficial, incluso sirven para cualquier otra herramienta de la familia xUnit. Sin embargo, hay ciertos detalles de implementación que pueden variar de unas versiones a otras.

Antes de nada, tenemos que asegurarnos de tener una carpeta con los archivos del framework. Lo más normal es tener una carpeta llamada "CPPUnit" donde estén todos los archivos. Además del propio código fuente, los más importantes son las librerías "culib.lib" y "TestRunnerDlg.lib", que necesitaremos enlazar con nuestra aplicación de pruebas y "TestRunnerDlg.dll" para ejecutar baterías de pruebas con interfaz gráfico de ventanas. Todos estos archivos los podéis encontrar en la carpeta "CPPUnit\bin". Si no tenemos estas librerías, podremos crearlas recompilando los proyectos en C++Builder 6. Para ello abrimos el grupo de proyectos "CPPUnit.bpg" y recompilamos todo con la opción de menú "Project - Build All Projects".

De todos los proyectos que vienen en el grupo de CPPUnit, vamos a utilizar sólo dos:

culib.lib: Se trata de una librería estática con el núcleo de pruebas de CPPUnit. Cualquier aplicación que quiera hacer pruebas aprovechando las facilidades de CPPUnit, debe enlazar con esta librería.

TestRunnerDlg.dll: es una librería dinámica que permite mostrar las pruebas de forma gráfica con una ventana de diálogo. Muestra un árbol donde se representan la estructura de colecciones y casos de pruebas, donde se pueden ir seleccionando las que queramos ejecutar. Además, muestra el progreso del proceso mientras ejecutamos la batería de pruebas.

Preparando el proyecto de pruebas

Una vez que hemos compilado el framework, tenemos que crear un nuevo proyecto desde C++Builder (File – New – Application) y eliminar el formulario principal que se incluye en la aplicación (Project - Remove from Project - Unit1.cpp). Después crearemos una nueva unidad (con sus archivos .cpp y .h) para codificar en ella los casos de pruebas.

Siguiendo con nuestro ejemplo, crearemos los archivos "DatosDiscoTest.h" y "DatosDiscoTest.cpp" para codificar los casos de prueba. Estos casos de prueba se van a encargar de probar todo lo relacionado con la lectura y grabación del dato, dentro de la clase "DatosDisco". En el archivo de cabecera "DatosDiscoTest.h" codificaremos la definición de clase que podéis ver en el siguiente listado:

```
#if !defined(_DATOS_DISCO_TEST_H_)
#define _DATOS_DISCO_TEST_H_

#include "TestCase.h"
#include "TestCaller.h"

#include "DatosDisco.h"


class DatosDiscoTest : public TestCase
{
public:
    DatosDiscoTest(std::string name);
    virtual ~DatosDiscoTest();

    void setUp();
    void tearDown();

    static Test *suite();
protected:
    void testLeer();
    void testGuardar();
private:
    DatosDisco  *fixture;
};

typedef TestCaller DatosDiscoTestCaller;

#endif
```

Dentro de la clase “DatosDiscoTest”, los métodos “setUp” y “tearDown” vienen heredados de la clase “TestCase”. El método “setUp” se llama automáticamente cuando iniciamos cada uno de los casos de prueba, y en este punto podemos inicializar cualquier recurso que necesitemos para hacer las pruebas. Por ejemplo, si nuestras pruebas tienen que ver con una base de datos, este suele ser el momento de realizar la conexión. El método “tearDown” es el complementario, y se ejecuta automáticamente cuando el caso de pruebas ha terminado.

El método estático “suite” retornará un nuevo objeto de tipo “Test” que representa a la colección de pruebas. Como la clase “Test” es una clase base, en realidad podremos retornar una instancia de cualquiera de sus hijos:

- **TestSuite**: un conjunto de casos de prueba, es decir: una colección de pruebas.
- **TestCase**: una prueba individual.

En la sección protected aparecen los métodos que implementan los casos de prueba. En nuestro caso tenemos dos métodos y cada uno de ellos ejecuta un único caso de prueba. Así, en el método “suite” que hemos visto antes, debemos indicar que nuestra colección de pruebas se compone de dos casos de pruebas: “testLeer” y “testGuardar”.

Por último, en la sección private tenemos un atributo con el objeto que vamos a probar. A este atributo se le conoce como fixture y se trata del objeto “sufridor” de

nuestras pruebas. En el método "setUp" crearemos la instancia, asignándola a este atributo, y en el método "tearDown" la liberaremos. De este modo conseguimos tener un objeto "fresco", recién creado, para cada una de nuestras pruebas. Si quieres saber porqué se diseñó CPPUnit de esta forma, puedes consultar esta entrada en el blog de Martin Fowler:

http://www.martinfowler.com/bliki/JunitNewInstance.html

La implementación de la clase es que podéis ver a continuación:

```
#include "DatosDiscoTest.h"

DatosDiscoTest::DatosDiscoTest(std::string name) : TestCase(name)
{
    fixture = NULL;
}

DatosDiscoTest::~DatosDiscoTest()
{
}

void DatosDiscoTest::setUp()
{
    fixture = new DatosDisco();
}

void DatosDiscoTest::tearDown()
{
    if (fixture != NULL) {
        delete fixture;
        fixture = NULL;
    }
}

Test* DatosDiscoTest::suite()
{
    TestSuite *suite;

    suite = new TestSuite("DatosDisco");

    suite->addTest( new DatosDiscoTestCaller("Leer",
                                &DatosDiscoTest::testLeer) );
    suite->addTest( new DatosDiscoTestCaller("Guardar",
                                &DatosDiscoTest::testGuardar) );

    return (suite);
}

void DatosDiscoTest::testLeer()
{
    // ejecución de las pruebas para leer
}

void DatosDiscoTest::testGuardar()
{
    // ejecución de las pruebas para guardar
}
```

La implementación es sencilla, al menos por ahora: en los métodos "setUp" y "tearDown" se crea y destruye el objeto de pruebas (fixture), para asegurarnos que está recién creado cada vez que ejecutemos la colección de pruebas, y en el método "suite" se crea un objeto "TestSuite" y se le añaden los casos pruebas que conformen esa colección, a través de la llamada al método "addTest".

Programando los casos de pruebas

Una vez que tenemos claro los que hay que probar, tenemos que ser capaces de programarlo. Para hacer esto vale todo. Lo más normal es lanzar métodos públicos de nuestro "fixture", y comprobar que el retorno es el esperado, y que los atributos del objeto han quedado con los valores correctos. En otras ocasiones, nos tenemos que apoyar en ficheros o tablas auxiliares (que contienen los datos correctos), funciones y métodos auxiliares (que podemos definir en esta misma clase), librerías de terceros, etc. Como decía: vale todo.

La forma de verificar los valores es a través de unos métodos especiales que comprueban una condición y generan una excepción si no es verdadera. Estos métodos se llaman "aserciones" y existen de distintos tipos:

- **assert(bool)**: genera una excepción si la condición no se cumple.
- **assertEquals(esperado, obtenido)**: genera una excepción si ambos valores son distintos, mostrando el valor esperado y el obtenido. Se pueden pasar parámetros de tipo "long", "double", "std::string" o "char *".
- **assertEquals(double esperado, double obtenido, double umbral)**: genera una excepción si la diferencia entre ambos valores es mayor al umbral, mostrando el valor esperado y el obtenido.
- **assertMessage(bool condición, char *mensaje)**: genera una excepción si la condición no se cumple, mostrando el mensaje indicado en el segundo parámetro.

Cada vez que una aserción no se cumple, se mostrará un error indicando que ese caso de prueba no se ha pasado, la línea, el fichero fuente, y la condición que no se ha cumplido en la aserción.

Hay dos razones por las que un caso se prueba no se da por válido:

- **Fallos**: una o más aserciones no se han pasado porque la condición no es verdadera. Esto denota que el código no pasa una de las comprobaciones que hemos impuesto.
- **Errores**: se ha generado algún tipo de excepción incontrolada en el código, como errores de acceso a memoria, errores de entrada/salida, excepciones del sistema operativo, etc. Esto significará que nuestro código contiene un error incontrolado.

En nuestro ejemplo, hemos codificado los dos casos de prueba del siguiente modo

leer: tenemos un fichero auxiliar donde están almacenados los datos. Además conocemos sus valores y sabemos que están correctamente grabados (a este fichero le llamaremos "patrón"). Se leerán los datos y se verificarán que lo leído es lo que realmente está almacenado en el fichero. Las verificaciones se hacen a través de las funciones "assert" y "assertEquals", para lanzarse la correspondiente excepción cuando la condición no se cumpla. La codificación de esto es la que muestra a continuación:

```
// Estos son los datos que sabemos que están correctamente guardados
// en el fichero patrón.
#define FICHERO_PATRON     "patron.dat"
#define NUMERO_PATRON      19
#define CADENA_PATRON      "texto de prueba del archivo patrón"

void DatosDiscoTest::testLeer()
{
    // se ejecuta la acción...
    assert( fixture->leer(FICHERO_PATRON) );

    // ...y se comprueban los resultados con aserciones
    assertEquals( NUMERO_PATRON, fixture->getDato()->numero );
    assert( 0 == strcmp(CADENA_PATRON, fixture->getDato()->cadena) );
}
```

guardar: se guarda un nuevo archivo con los mismos valores que en el patrón. Después se leen ambos archivos y se comparan sus contenidos, a través de la función auxiliar "SonFicherosIguales", tal y como vemos en el siguiente listado. Deben ser iguales para considerar que la clase guarda correctamente la información.

```
void DatosDiscoTest::testGuardar()
{
    const char* FICHERO_TMP = "copia.tmp";

    DATOS    d;

    d.numero = NUMERO_PATRON;
    strcpy(d.cadena, CADENA_PATRON);

    // se ejecuta la acción...
    fixture->setDato(&d);
    fixture->guardar(FICHERO_TMP);

    // ...y se comprueban los resultados
    // El contenido del temporal debe ser el mismo que el del patrón
    // Para ello utilizo la aserción "assertEqualFiles"
    assert( sonFicherosIguales(FICHERO_TMP, FICHERO_PATRON) );
}
```

Atención
Aunque por simplicidad he utilizado estos ejemplos de pruebas unitarias, hay que evitar en lo posible que los test dependan de archivos auxiliares, tablas o registros en base de datos o cualquier otro recurso a parte de propio código fuente.
Los test deberían ejecutarse simplemente compilando el código, sin necesitar de otros archivos.

La excepción es la que confirma la regla

La norma general es que una excepción denote un fallo en la prueba unitaria, aunque existe un caso que es "la excepción que confirma la regla" (y nunca mejor dicho): ciertas librerías lanzan las excepciones soportadas por ANSI-C++ para avisarnos de algún error. En estos casos, debemos capturar la excepción, ya que se trata del comportamiento esperado, y no un error. Vamos a explicarlo con un ejemplo: supongamos que el método "setDato" de la clase "DatosDisco" comprueba la validez del puntero que se pasa, y lanza una excepción si este puntero no es correcto (por ejemplo si es nulo). Este sería el comportamiento lógico, y un programador debería controlar estas situaciones. Podemos escribir un caso de prueba que verifique este comportamiento, considerando la excepción como el caso correcto, y la ausencia de excepción como un error, ya que la clase debería haber lanzado la excepción que esperábamos. Para estos casos se debe hacer uso de la instrucción try-catch, con la que capturamos una excepción. La implementación de este caso de prueba sería la que se muestra a continuación:

```
void DatosDiscoTest::testDatoNulo()
{
    try {
        fixture->setDato(NULL);
        // si pasa por aquí, entonces error
        assertMessage(false, "Error en setDato(): no ha saltado excepción.");
    }
    catch(...) {
    }
}
```

Como podéis ver, si se produce alguna excepción dentro de la función "setDato", el flujo de ejecución saltará dentro del "catch", donde la excepción se dará por anulada, continuando la ejecución como si no hubiera pasado nada. Si por algún error, la excepción no se produce (aunque debería), la ejecución seguirá su curso normal, pasando por el "assert(false)", donde se levantará una excepción de error, indicando que algo no está funcionando bien, ya que el método "setDato" debería haber lanzado una excepción.

Lanzando el interfaz de pruebas

Por último nos queda ver cómo hacer que aparezca en pantalla el interfaz gráfico gestionado por la librería "TestRunnerDlg.dll".

Para ello debemos abrir el código fuente del proyecto (Project – View Source) y codificar lo que veis a continuación:

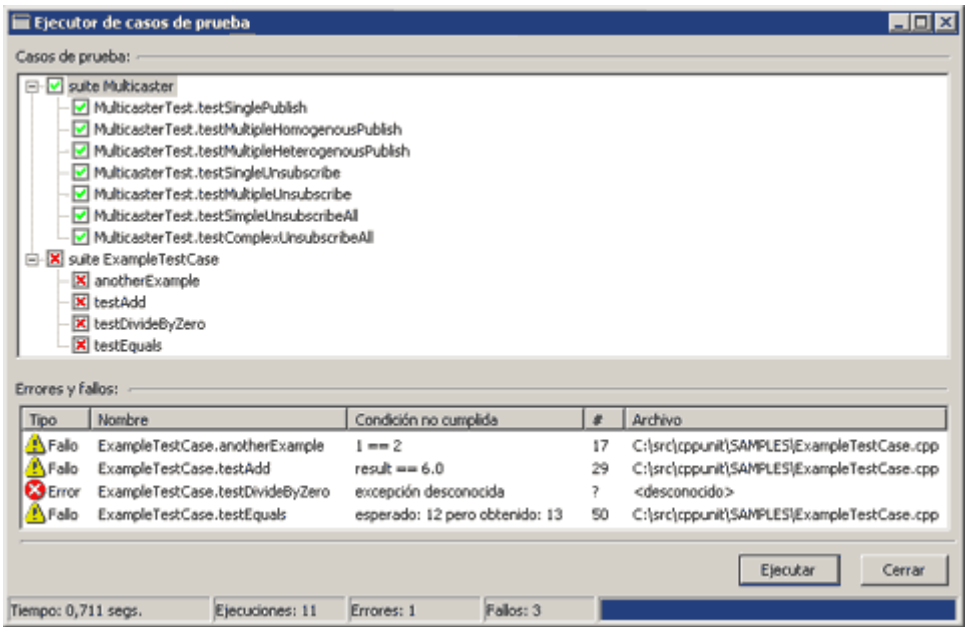
```
#include
#pragma hdrstop

#include "ITestRunner.h"
#include "DatosDiscoTest.h"

WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        ITestRunner     runner;

        runner.addTest( DatosDiscoTest::suite() );

        // aquí añadir cualquier otra colección de pruebas

        runner.run();
    }
    catch (Exception &exception) {
        Application->ShowException(&exception);
    }
    catch (...)
    {
        try {
            throw Exception("");
        }
        catch (Exception &exception) {
            Application->ShowException(&exception);
        }
    }
    return (0);
}
```

La mayoría de este código nos vienen dado por el propio C++Builder, a excepción de los includes de "ITestRunner.h" y "DatosDiscoTest.h" y las líneas contenidas dentro del primer try.

Cada vez que queramos añadir una nueva colección de pruebas, debemos incluir el archivo de cabecera donde lo hayamos codificado, y añadir la línea:

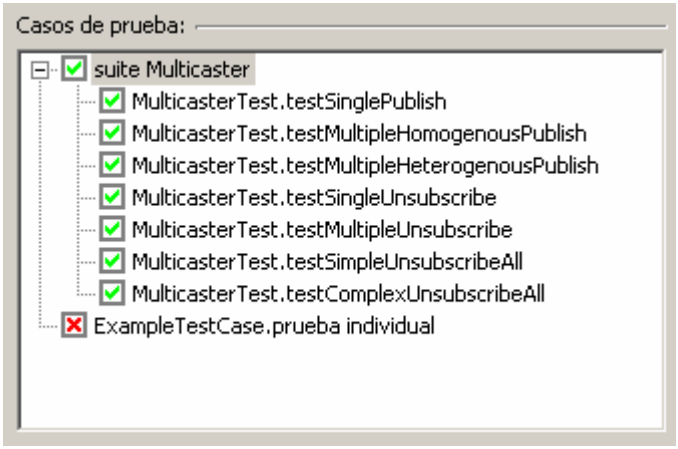
```
runner.addTest(ClaseDeCasosDePrueba::suite());
```

Si compilamos y ejecutamos el proyecto de pruebas podemos ver el interfaz gráfico, donde se presenta un árbol con las colecciones de pruebas en el primer nivel, y los casos de prueba contenidos en cada colección, en el segundo nivel, tal y como podemos ver a continuación:



Una vez ejecutado, aparecerán en verde aquellos casos de prueba que se han ejecutado satisfactoriamente, y en rojo aquellos que no se consideran correctos.
En la parte inferior, puede verse una descripción de los errores que se han producido, mostrando el tipo de error (fallo de aserción o error de ejecución), el nombre del caso de prueba donde se ha generado el error, la condición que no se ha cumplido y el número de línea y archivo fuente donde se ha producido el error.

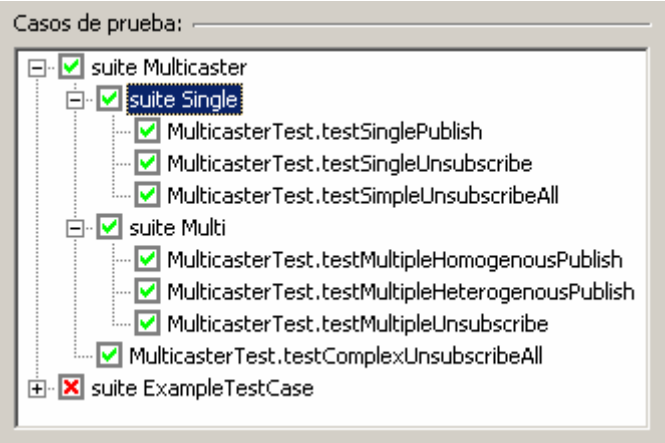
Baterías de pruebas anidadas



La imagen anterior muestra la información que nos da CPPUnit de una típica batería de pruebas: varias colecciones, formada cada una de ellas por varios casos de pruebas.

Sin embargo, aunque el ejemplo que aquí hemos desarrollado es el uso típico, CPPUnit permite otros tipos de baterías más complejas.
Por ejemplo, podemos conseguir que no existan colecciones, y mostrar los casos de prueba directamente en el primer nivel, como se muestra en la imagen de la derecha.

Y también podemos hacer que una colección de pruebas contenga a su vez a otra colección, pudiendo así crear estructuras de pruebas anidadas:



Para conseguir esto, debemos cambiar el objeto que retornaremos en el método suite.

Recordemos su prototipo:

```
Test* DatosDiscoTest::suite()
```

Podemos ver que el método suite acepta que retornemos un objeto de tipo Test o cualquier descendiente.

Si nos fijarnos bien en la jerarquía de clases de CPPUnit, veremos que tenemos distintos objetos que podríamos retornar desde el método suite.

Vamos a ver algunas de las clases que aparecen en el diagrama y cómo podemos utilizarlas de formas alternativas para modificar la estructura de las pruebas

- La clase **TestCaller** ya la hemos utilizado para crear un test individual y añadirlo a una colección de pruebas. Con esta clase también conseguiremos ejecutar una prueba individual simplemente dando su nombre y el método donde hemos codificado la prueba. Con esto podremos evitar el uso de las colecciones y mostrar pruebas individuales en el primer nivel de nuestra batería de pruebas:

```
Test* DatosDiscoTest::suite()
{
    TestCaller *caller;

    caller = new DatosDiscoTestCaller("testLeer",
                        &DatosDiscoTest::testLeer) );

    return (caller);
}
```

- La clase **TestCase** también la hemos utilizado, pero para un uso distinto: como contenedor de todos los métodos que implementan las pruebas. Existe otro uso, que nos permite ejecutar una prueba individual utilizando un TestCase. Basta con sobrescribir el método runTest y codificar ahí nuestra prueba, tal y como veremos en el siguiente código. Esto nos sirve para el mismo propósito que el código anterior, pero ahorrándonos el uso del TestCaller.

```
// en el archivo de cabecera
class DatosDiscoTest : public TestCase
{
public:
    ...
protected:
    // en este método codificamos la prueba individual
    virtual void runTest(TestResult* r);
};

// en la implementación
Test* DatosDiscoTest::suite()
{
    TestCase *prueba;

    prueba = new TestCase("DatosDiscoTest");

    return (prueba);
}
```

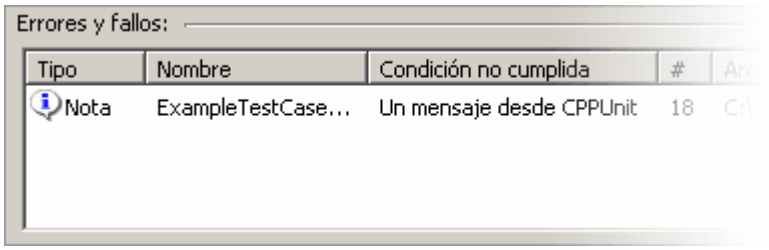
- Y por último, la clase **TestSuite** es la que hemos estado utilizando hasta ahora. Con ella podemos crear una colección de pruebas, configurando previamente el contenido de la colección.
  A la hora de añadir el contenido de la colección, debemos usar el método "addTest" de TestSuite, y podremos utilizar como parámetro también cualquier objeto descendiente de Test, tal y como vemos en su prototipo:

```
void addTest(Test *test);
```

Esto nos permite añadir tanto un TestCaller (como hemos hecho hasta ahora), un TestCase (si implementamos el método runTest, como ya hemos explicado) o incluso otro TestSuite (porque también es un descendiente de Test). Esto nos permite crear jerarquías de clases, tal y como hemos explicado ya. El código sería algo parecido a lo siguiente:

```
Test* DatosDiscoTest::suite()
{
    TestSuite *suite;

    suite = new TestSuite("DatosDisco");
    suite->addTest( OtroTest::suite() );

    return (suite);
}
```

Mensajes durante las pruebas



Hay ciertas pruebas en las que no es fácil conseguir una condición para escribirla en un `assert`. Puede ser porque la condición depende de factores externos al test, o porque sea una condición difícil de calcular con una expresión booleana (por ejemplo, aquellas en las que interviene la intuición, o la estadística). En esos casos, es habitual que necesitemos mostrar durante las pruebas algún mensaje, para que, en el momento de su ejecución, podamos determinar si el test está siendo correcto o no.

La llamada, desde cualquier método dentro de un caso de prueba, es bien sencilla:

```
showMessage("Mensaje que queremos que se muestre");
```

Para estos casos (y solo para estos), existe una llamada que nos puede ayudar: showMessage, que nos permite mostrar un texto en la zona de mensajes de la ventana de CPPUnit, tal y como aparece en la imagen de la derecha.

Atención
Los mensajes en las pruebas, aunque son una característica que puede resultar útil, no están para sustituir las llamadas a los `assert`.
Utiliza solo los mensajes cuando necesites ver algún dato durante la ejecución de un test, pero nunca para informar si el test ha pasado o no.

Cronómetro

Una de las situaciones típicas en la que necesitas mostrar mensajes, es cuando estás midiendo el rendimiento de una tarea. CPPUnit te puede ayudar si quieres hacer una medida inicial del rendimiento de ciertas llamadas, aunque si lo que quieres es un análisis de rendimiento más exhaustivo, te recomiendo que utilices herramientas específicas de profiling como AQTime.

Para tomar tiempos de ejecución utilizando CPPUnit, se proporcionan cuatro métodos a los que puedes llamar durante la ejecución de tus casos de prueba:

```
void startTiming(bool pausado);
void pauseTiming();
void resumeTiming();
void stopTiming(char* mensaje);
```

Estos métodos permiten iniciar, pausar, reanudar o parar el cronómetro interno de CPPUnit. En el momento de parar el cronómetro, se mostrará un mensaje (utilizando el método `showMessage`) informando del tiempo que se ha tardado en ejecutar la operación.

Pongamos un pequeño ejemplo: tratamos de medir el tiempo que tardarían 1.000 lecturas consecutivas utilizando la clase "DatosDisco", así que sencillamente haríamos un test como el siguiente:

```
void DatosDiscoTest::testRendimientoLectura()
{
    const int ITERACIONES = 1000;

    int i;

    // se arranca el cronómetro, pero pausado
    startTiming(true);

    i = 0;
    while (i++ < ITERACIONES)
    {
        // se reanuda el cronómetro
        resumeTiming();

        // se ejecuta la acción...
        fixture->leer(FICHERO_PATRON);

        // se pausa el cronómetro
        pauseTiming();
    }

    // se para el cronómetro para mostrar los resultados
    stopTiming("Tiempo de ejecucución");
}
```

Al terminar la ejecución, aparecerá un mensaje indicando el texto pasado a `stopTiming` y el tiempo de ejecución en formato hh:mm:ss,mmm.

El código fuente

CPPUnit para C++Builder 6:
http://users.servicios.retecal.es/sapivi/src/CPPUnitBCB6.zip

Versión oficial de CPPUnit:
http://cppunit.sourceforge.net/

Autor: JM

Este artículo apareció publicado por primera vez en el número 5 de la revista Todo Programación, editada por Studio Press, S.L. y se reproduce aquí con la debida autorización.