`BUnit.h`: Un módulo C++ simple para aprender prueba unitaria de programas

Adolfo Di Mare
Universidad de Costa Rica
Escuela de Ciencias de la
Computación e Informática
+1 506 2207-4020
adolfo.dimare@ecci.ucr.ac.cr

ABSTRACT
It is pleaded for the use of unit testing program tools. A scheme for program implementation in which both the specification and the unit test for each piece of software is developed before coding algorithms. The experience on using this approach with programming students is reported.

RESUMEN
Se promociona el uso de herramientas para la prueba unitaria de programas. Se propone un esquema de implementación de programas en el que tanto la especificación y la prueba de cada pieza de software se desarrolla antes de la codificación de los algoritmos. Se reporta la experiencia del uso de este enfoque de construcción de programas en estudiantes de programación.

Categorías y Descriptores
D.2.5 [**Testing and Debugging**]: Testing tools. D.2.1 [**Requirements/Specifications**]: Tools.

Términos Generales
Algorithms, Documentation, Design, Economics.

Keywords
Prueba unitaria de programas, unit testing, ingenieria de sistemas, software engineering.

1. INTRODUCCION
Mejorar la productividad del programador es esencial para reducir el costo de construir sistemas de computación, pues la mayor parte del presupuesto se invierte pagándoles. Por eso, el entrenamiento de programadores enfatiza lograr que utilicen herramientas sofisticadas que les ayuden a aumentar su productividad. Aunque ya se han logrando mejoras en la productividad usando formas innovadoras de organización, como las metodologías "ágiles" de "programación extrema" (XP: eXtreme Programming) [2] o el uso de los lenguajes de cuarta generación [6], todavía existe la expectativa de reducir más el costo de la programación.

En este artículo se describe la herramienta BUnit, un módulo [B]ásico para prueba [unit]aria de programas, que sirve para lograr que un programador C++ adquiera la buena costumbre de diseñar, especificar y probar antes de codificar los algoritmos de un programa. La cualidad más importante del componente BUnit es su sencillez, que lo hace ideal para mejorar la construcción de programas; está diseñado para que los principiantes aprendan a hacer prueba unitaria de programas y es un paso hacia el uso de las otras herramientas más sofisticadas que usa el programador en su práctica profesional. Aquí se argumenta que siempre es necesario:

- Construir la especificación antes de la implementación.
- Incorporar los datos de prueba como parte de la especificación de cualquier módulo.

2. DOXYGEN ES PARTE DE LA SOLUCIÓN
Así como las bailarinas son felices cuando danzan, los programadores obtienen su gratificación cuando programan. Escriben especificaciones obligados, pero prefieren invertir su tiempo codificando (muchas veces están anuentes a escribir una corta descripción de lo que hace cada módulo, aunque no quede redactada de la mejor manera). Debido a que es necesario que el programa funcione para que el cliente pague la siguiente cuota pactada, el énfasis del entrenamiento del programador está en lograr que implemente pronto el programa. Por eso pierden importancia la documentación y, en especial, la especificación de cada uno de los módulos de un sistema. Es particularmente difícil que los programadores aprendan a especificar sus módulos, pues pocos tienen la paciencia de redactar con cuidado sus ideas: son "ingenieros", no "poetas". Herramientas como Doxygen [4] sirven para mejorar significativamente la situación actual. En la Universidad de Costa Rica se usa este generador de documentación, desarrollado por Dimitri van Heesch [7], porque es una herramienta que cubre un espectro más amplio que el de herramientas como "JavaDoc", pues sirve generar la documentación para los lenguajes más populares: C++, Java, C#, Visual Basic, SQL, etc.

El uso de herramientas para probar programas no representa un enfoque revolucionario, pues no propone un nuevo paradigma de programación, y más bien complementa los esquemas formales usados para construir programas, en lugar de modificarlos o anularlos. Por eso es una buena idea usar la prueba de programas como una parte adicional de cualquier pieza de software. Aquí se propone mejorar la construcción de sistemas incorporando la prueba unitaria del módulo como parte de su especificación.

Una barrera que es necesario franquear es lograr permear a los profesores y directores de proyecto para que acepten y adopten la buen práctica de escribir programas construyendo las especificaciones y los datos de prueba antes que los algoritmos. Asimilar las técnicas de prueba unitaria de programas es más fácil si se usa una herramienta que usan es adecuada y sencilla como la pareja Doxygen-BUnit.

En la Universidad de Costa Rica el uso de BUnit y Doxygen se ha logrado acostumbrando a los profesores a que sus alumnos entreguen sus trabajos programados usando estas herramientas que tienen la cualidad de que son fáciles de usar. En un ambiente de trabajo de empresa en que la prueba unitaria de programas no es una práctica cotidiana es posible lograr lo mismo.

3. ARQUITECTURA DE BUNIT

Chuck Allison implementó un pequeño módulo para probar programas C++ [1] que disminuye el costo de aprender a usar paquetes de calidad profesional como CppUnit. Sin embargo, usar herramientas similares a CppUnit requiere del programador un entendimiento profundo de cómo están construidas, y no sirve para mejorar la especificación de los módulos. El módulo C++ BUnit que se presenta aquí es más simple que el propuesto por Allison, y tiene la ventaja de que se puede usar con facilidad para mejorar la documentación.

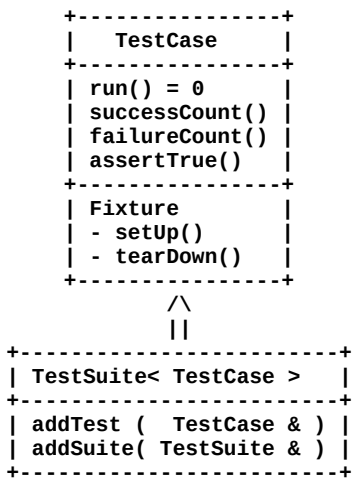

Figura 1. Clases de BUnit.

Como se muestra en la Figura 1, BUnit está constituida por 2 clases. La clase base TestCase que contiene la implementación de la prueba y también la colección en donde están almacenados los resultados de ejecutar la prueba. La clase emplantillada TestSuite<> contiene varias pruebas y, por comodidad, está derivada de TestCase (se han usado los nombres en inglés para mantener la compatibilidad con otras herramientas de la familia "xUnit").

La cualidad que distingue a BUnit es que es muy simple, tanto que su implementación completa reside en un único archivo de encabezado: BUnit es una de las más simples herramientas de la familia "xUnit", que es el nombre acuñado a partir de la herramienta Java originaria: JUnit (varios de los miembros de esta familia de herramientas para prueba unitaria de programas están descritas en [8]). Si la herramienta es simple el programador opondrá menos objeciones para usarla y también servirá para integrar al aprendizaje de manera natural la programación el uso de la prueba unitaria de programas. Como BUnit es simple también es más fácil de documentar. Un diseño simple contiene sólo las cualidades más importantes de la herramienta de manera que sea el uso de otras herramientas más complicadas en donde se incorporen las opciones y facilidades más sofisticadas.

Herramientas más complejas incorporan elementos gráficos que ayudan al programador, como por ejemplo el uso de una "barra de progreso" que indique cuánto falta para terminar, lo que requiere mucha maquinaria programática que es necesario tomar en cuenta y que por lo tanto incrementa la complejidad de la herramienta. Otra importante cualidad de BUnit es que se usa de una manera similar a JUnit, la herramienta de prueba unitaria que la mayor parte de los programadores Java conocen, lo que facilita el entrenamiento de nuevos programadores en herramientas más sofisticadas, quienes eventualmente deben hacer esa transición. BUnit se parece a JUnit para facilitar la transición entre ambas herramientas. Para implementar una prueba unitaria con BUnit basta seguir estos pasos:

1. Agregrar `#include "BUnit.h"` en el programa.
2. Derivar de TestCase una clase para implementar las pruebas.
3. Implementar en el método run() las pruebas invocando assertTrue().
4. Usar el método run() para ejecutar las pruebas.

```
#include "BUnit.h" // 1. Agregrar #include "BUnit.h"

/// Ejemplo mínimo de uso de \c BUnit.
class test0 : public TestCase {// #2. Derivar de TestCase
public:
    bool run() {
        assertTrue(1+1 == 3); // #3 Invocar assertTrue()
        return wasSuccessful();
    }
};

#include <iostream> // std::cout

/// Programa principal que ejecuta la prueba.
int main() {
    test0 test0Instance;
    test0Instance.run(); // #4 run(): Ejecuta la pruebas
    if ( ! test0_instance.wasSuccessful() ) {
        std::cout << test0_instance.report();
    }
    return 0;
}
```

Figura 2. El programa de prueba test0.cpp

El programa de la Figura 2 muestra cómo usar el método assertTrue() para constatar que la expresión booleana que recibe como argumento es verdadera. En este caso, debido a que la suma 1+1 es diferente de 3, el total de pruebas erróneas queda aumentado en 1, hecho que queda grabado en la salida estándar std::cout cuando alguna prueba unitaria no tiene éxito; en el caso contrario este programa no grabaría nada. El resultado de la ejecución muestra que en el renglón (7) la prueba falló:

```
TestCase [class test0] (OK: 0) (FAIL: 1)
=\_fail: 1 + 1 == 3
=/ (6) X:\DIR\SubDir\test0.cpp
```

Como práctica básica de programación es saludable que la clase de prueba sea una clase amiga (*friend*) de la clase con que se trabaja. Por ejemplo, en declaración de la clase "Machine" estaría declarada como "*friend*" la clase "test_Machine". Esto facilita la escritura de casos de prueba de caja blanca, y evita la proliferación de declaraciones de clases amigas para la clase "Machine".

4. ESPECIFICACIÓN POR MEDIO DE PRUEBA DE PROGRAMAS

Dice un famoso adagio que más vale una imagen que mil palabras. En el contexto de la construcción de programas lo mismo puede decirse de los ejemplos: "más vale un ejemplo que mil palabras":

```
bool Graph::connected(
    const std::string & src,
    const std::string & dst,
    std::list< std::string > & C
);
```

El prototipo de una función es su mínima especificación. Aunque se puede intuir qué hace connected(), es mejor que la especificación incluya un buen ejemplo de uso:

Determina si existe un camino en el grafo comenzando en "src" y terminando en "dst".
* Si src == dst retorna "true" (un vértice siempre está conectado consigo mismo).
* Retorna "true" cuando el camino existe, y "false" en caso contrario.
* La lista "C" contiene la secuencia de nodos del camino.
* Si no hay camino, la lista "C" queda vacía.

```
bool Graph::connected(
    const std::string & src,
    const std::string & dst,
    std::list< std::string > & C
);
{{ // test::diagram()
     A(1)         C(1)        O(1)---->O(2)
    /    \       /    \       /|\       |
   /      \     /      \       |        |
F->--A(2)-->-B->         ->D   |        |
   \      /     \      /       |        |
    \    /       \    /        |       \|/
     A(3)         C(2)        O(4)<----O(3)
}}
{{ // test::connected()
    std::list< std::string > C; // camino en el grafo
    std::list< std::string >::iterator it;

    // no existe el vértice
    assertTrue( ! G.connected( "???" , "???",  C ) );
   // grafo no conexo
    assertTrue( ! G.connected(  "F" ,  "O(4)", C ) );

    assertTrue( C.size() == 0);

    // el grafo es dirigido
    assertTrue( ! G.connected( "D" , "F"   , C ) );
    assertTrue( C.size() == 0);

    // si está conectado
    assertTrue( G.connected( "F" , "F", C ) );

    // porque ya está ahí
    assertTrue( C.size() == 1 && C.front() == "F" );

    assertTrue( ! G.connected( "D", "A(2)" , C ) );
    assertTrue( C.size() == 0 );
    assertTrue( G.connected( "A(2)" , "D", C ) );
    assertTrue( C.size() == 4 );
    assertTrue( C.front() == "A(2)" && C.back() == "D" );
    it = C.begin(); it++;
    assertTrue( *it == "B" ); // 2do nodo en el camino
    it++;
    assertTrue( *it == "C(1)" || *it == "C(2)" );
    // 3er nodo en el camino
}}
```

Figura 3. Especificación completa.

En la Figura 3 los datos de prueba reales se han usado para complementar la especificación del método connected(). Los datos de prueba Bunit están rodeados por un bloque de corchetes dobles "{{ ... }}" para que el generador de la documentación (que en este caso es Doxygen) pueda identificar el bloque de código que hay que agregar a la especificación. Por supuesto, ese bloque de código ha sido extraído del programa de prueba y por eso es un ejemplo real, que ya ha pasado el tamizaje del compilador, y que puede ser ejecutado efectivamente.

5. FORMATO DE LA DOCUMENTACIÓN DOXYGEN

Es natural que los detalles de uso de una herramienta de documentación determinen la forma en que se llega a lograr incorporar código de prueba como parte de la especificación. En el caso de Doxygen, esta herramienta incluye el comando "\dontinclude" que sirve para copiar en la documentación final un pedazo obtenido de otro archivo. Para determinar adonde comienza y termina el bloque de código hay que usar 2 hileras que lo identifiquen. Para la Figura 3 se usaron las hileras "test::diagram()" y "}}", que marcan el principio y final del bloque de código que se quiere incluir como parte de una especificación.

```
/** Determina si existe un camino en el grafo comenzando
    en \c "src" y terminando en \c "dst".
    - Si <code> src == dst </code> retorna \c "true" (un
      vértice siempre está conectado consigo mismo).
    - Retorna \c "true" cuando el camino existe, y
      \c "false" en caso contrario.
    - La lista \c "C" contiene la secuencia de nodos del
      camino.
    - Si no hay camino, la lista \c "C" queda vacía.
    \dontinclude test_Graph.cpp
    \skipline    test::diagram()
    \until       }}
    \skipline    test::connected()
    \until       }}
    \see         test_Graph::test_connected()
*/
bool Graph::connected(
    const std::string & src ,
    const std::string & dst ,
    std::list< std::string > & C
) {
// ... implementación
}
```

Figura 4: Comentarios Doxygen que resultan en la especificación de la Figura 3

En la Figura 8 está la codificación de la especificación en el programa fuente. Con el comando Doxygen "\dontinclude" se define cuál es el archivo del que se tomará texto para agregarlo a la documentación; en este caso el archivo que contiene las pruebas para la clase se llama test_Graph.cpp. Con los comandos "\skipline" y "\until" se define la parte del archivo mencionado en "\dontinclude" que será incorporada en la documentación final. Para definir el primer renglón del bloque de código hay que identificarlo con una hilera; para obtenerla, el truco usado es concatenar la palabra "test" con el nombre del método a prueba, "connected()" en este caso, para obtener la hilera de identificación completa "test::connected()".

Con el fin de que quien usa la documentación pueda examinar con comodidad todas las pruebas, conviene también incluir el nombre del método que las contiene, lo que se logra con el comando "\see" que le indica a Doxygen que genere un salto hacia la documentación del método referenciado. Lo usual en estos días es navegar por la documentación interactivamente, pero si la documentación está siendo generada para ser impresa en muchos casos será mejor dejar por fuera esta referencia.

Algunos detalles menores que también hay que tomar en cuenta son los siguientes. Es necesario especificarle a Doxygen adónde están los archivos mencionados en el comando "\dontinclude": esto se logra incluyendo en renglón "EXAMPLE_PATH" del archivo de configuración. Además, con frecuencia conviene incluir la opción "JAVADOC_AUTOBRIEF" que facilita la escritura de la documentación corta de cada ítem.

Si en algún caso el archivo que se está documentando es el mismo del hay que extraer código, caso que ocurre cuando el archivo mencionado en el renglón "INPUT" es también el utilizado para extraer documentación con el comando "\dontinclude" es importante que el texto de ejemplo aparezca después del lugar en que aparece la especificación, pues de lo contrario el comando "\dontinclude" no agregaría el texto de ejemplo a la documentación, pues la hilera que marca el principio del bloque a incluir aparece antes.

```
/// Datos de prueba para los constructores
/// de la clase \c TestCase.
void test_BUnit::test_constructor() {
    {{  // test::constructor()
        test_BUnit thisTest;
        assertTrue(
            string::npos !=
            thisTest.getName().find( "test_BUnit" )
        );
        assertTrue( thisTest.failureCount() == 0 );
        assertTrue( thisTest.countTestCases() == 1 );
        assertTrue( thisTest.successCount() == 0 );
        assertTrue( thisTest.runCount() == 0 );
        assertTrue( thisTest. failureString() == "" );
    }}
    {   // Resto de las pruebas
        test_BUnit thisTest;
        assertTrue( thisTest.m_pass == 0 );
        assertTrue( thisTest.m_failure == 0 );
        assertTrue( thisTest.m_name == 0 );
        assertTrue( thisTest.m_failureList.empty() );
    }
}
```

Figura 5: Implementación de las pruebas para el constructor de la clase TestCase

En la Figura 5 se muestra cómo queda una prueba completa en el archivo de pruebas. Al principio está el ejemplo que aparecerá como parte de la documentación envuelto en el un bloque de corchetes dobles "{{ ... }}". Luego aparece el resto de las pruebas, cuya existencia se justifica porque la prueba unitaria de programas debe ser completa y, a veces, exhaustiva, pero ese código agrega poco a la documentación. Como lo usual es que las clase de prueba pueda ver lo privado de la clase, en este caso el resto de la prueba "se le mete al *Rep*"de la clase [3]. Todos los fuentes de BUnit están aquí:

http://www.di-mare.com/adolfo/p/BUnit/BUnit.zip

6. EXPERIENCIA CON ESTUDIANTES

En contraste con lo que ocurre con la pareja Doxygen-BUnit, la mayoría de las herramientas para prueba de programas son complicadas de entender y de usar. Debido a que han sido diseñadas para ambientes de trabajo especiales, orientados al uso de metodologías de programación "ágiles" como XP, es poco frecuente encontrarles uso o espacio en ambientes académicos. En consecuencia, los alumnos pasan por la universidad acostumbrados a "probar mañana y nunca hoy", lo que efectivamente resulta en un significativo problema de formación. Por eso, es necesario contar con una herramienta simple que le sirva a los alumnos, de manera que al usarla logren aumentar su productividad al mismo tiempo que incorporan en su diario quehacer la disciplina de primero probar y luego codificar.

Los estudiantes que han sido expuestos a la pareja Doxygen-BUnit con frecuencia no lo notan. En lugar de recalcarles su obligación de incluir prueba unitaria de programas, más bien se les menciona que deben completar con ejemplos su documentación. Además, en aquellos casos en que un proyecto consiste en implementar una parte de una clase, o un grupo de rutinas o métodos, la definición del problema incluye los casos de prueba como documentación, lo que también usan al construir su solución al proyecto. Debido a que documentar es "feo" para los programadores, hacer casos de prueba se transforma en algo "bonito", pues ese trabajo es percibido como programación y no como documentación. Esto ayuda mucho a que el trabajo de realizar pruebas sea realmente provechoso y desafiante, y no aburrido por ser obligatorio. En otras palabras, el uso de estas 2 herramientas resulta natural a los estudiantes quienes no gastan esfuerzo en luchar contra su uso sino que, más bien, de manera natural lo incorporan a su trabajo. En muchas ocasiones es útil no incluir palabras elevadas, como "prueba unitaria", "programación extrema" o "especificación", de manera que los muchachos no tienen que lidiar con conceptos abstractos sino que más bien es con la práctica que llegan a incorporar la buena disciplina de documentar módulos usando prueba unitaria de programas.

7. DETALLES DE IMPLEMENTACIÓN.

El verbo assertTrue() está implementado como una macro C++. Mediante el uso de las macros predefinidas del compilador __LINE__ y __FILE__, junto con el operador # (*stringify*), que permite obtener el texto de la prueba que se usa como argumento en assertTrue(), se logra crear la hilera de falla junto con la mención del renglón en donde se produjo la falla. Esta implementación parece poco elegante, pues los programadores C++ evitan siempre que pueden el uso de macros, pero parece inevitable para implementar BUnit.

La macro assertTrue() sirve para invocar al método testThis() que se encarga de realizar la prueba y de registrar su éxito o fracaso. Como assertTrue() es una macro, genera una hilera que contiene la condición boolean que hay que evaluar testThis() y además incluye la indicación del renglón y el archivo en donde está la invocación, lo que sirve después para reportar las pruebas que fallan.

```
bool run() {
    for ( int i=0; i<N; ++i ) {
        for ( int j=0; j<N; ++j ) {
            std::string err = "m_Matrix";
            err += '[' + TestCase::toString(i) + ']';
            err += '[' + TestCase::toString(j) + ']';
            err += " == 0";
            assertTrue_Msg( err , m_Matrix[i][j] == 0 );
        }
    }
    return wasSuccessful();
}
```

Figura 6: Afinamiento de los mensaje por medio de assertTrue_Msg()

En la Figura 6 se muestra cómo puede el programador cliente lograr que el mensaje de falla incluya información adicional que con el uso de assertTrue() se perdería. En este caso, en lugar de obtener el mensaje de falla genérico m_Matrix[i][j] == 0, que indica que una prueba no tuvo éxito posición "i-j" de la matriz, se obtiene un mensaje más descriptivo que incluye la

referencia exacta de la casilla en donde la prueba no tuvo éxito (en este caso 317-254):

```
=\_fail: m_Matrix[317][254] == 0
=/ (31) X:\DIR\SubDir\test1.cpp
```

Para mejorar la posibilidad de que quien aprende con BUnit pueda luego usar también con comodidad tanto JUnit como CppUnit, en la implementación se han incluido varias versiones del verbo assertTrue() que facilitan su uso. Algunas de las más importantes son éstas:

assertTrue(condition)
JUnit → assertTrue(condition)
CppUnit → CPPUNIT_ASSERT(condition)
assertTrue_Msg(message, condition)
JUnit → assertTrue(msg, condition)
CppUnit → CPPUNIT_ASSERT_MESSAGE(msg, condition)
assertEquals(expected, actual)
JUnit → assertEquals(expected, actual)
CppUnit → CPPUNIT_ASSERT_EQUAL(expected, actual)
assertFalse(condition)
JUnit → assertFalse(condition)
CppUnit → CPPUNIT_ASSERT(!(condition))
fail_Msg(message)
JUnit → fail(message) y también fail()
CppUnit → CPPUNIT_FAIL(message)

Lograr que toda la implementación de BUnit resida en un único archivo de encabezado requirió de varias contorsiones programáticas. Debido a que toda la implementación está contenida en el archivo "BUnit.h", lo único que un programador necesita para construir su prueba unitaria de programas es agregar lo con la directiva:

```
#include "BUnit.h"
```

Lo usual al implementar una clase C++ es usar 2 archivos: el de encabezado de extensión ".h" y el de implementación, de extensión ".cpp". Sin embargo, cuando se usan plantillas, toda la implementación debe estar en el archivo de encabezado. Los compiladores modernos se encargan de resolver los conflictos que ocurren cuando el mismo archivo de encabezado queda, completo, en 2 o más archivos de implementación. Esta ayuda adicional del compilador es lo que motivó a implementar la clase "TestSuite" usando plantillas. Para hacer más fácil la lectura del código se usó el identificador "TestCase" al implementar la plantilla "TestSuite"; esto obligó a una contorsión extraña, cual es crear un tipo sinónimo de "TestCase". De esta forma, es posible informarle al compilador cuál es la clase de la que deriva "TestSuite" al mismo tiempo que se usa es clase como nombre del parámetro de la plantilla.

Debido a que la mayor parte de los métodos de la clase "TestCase" son muy simples, casi todos fueron implementados como métodos "inline"; por eso no es necesario implementar "TestCase" con plantillas.

Para almacenar las pruebas que no tuvieron éxito se usa una lista, la que contiene las fallas almacenadas en valores "TestCaseFailure". Esta clase presenta una particularidad adicional, pues en algunos casos contiene hileras que deben ser destruidas por el destructor de la case. Lo usual es que las hileras de falla sean generadas por el compilador como hileras constantes, por lo que destruirlas es una equivocación. En otros casos, ocurre que la hilera se obtiene después de hacer varias operaciones con objetos std::string, en cuyo caso el valor almacenado sí debe ser destruido. Para manejar este caso particular, en la clase "TestCaseFailure" se guarda una indicación de cuándo es necesario destruir la hilera asociada a una falla: esto explica por qué es necesario el parámetro "must_copy" del método "testThis()".

Al escribir las pruebas muchas veces aparece la tentación de imprimir algo. Martin Fowler dijo una vez [6]: "Siempre que tenga la tentación de imprimir algo o de usar una expresión para el depurador simbólico, más bien escriba un caso de prueba". De esa manera se logra aprovechar el esfuerzo realizado para perpetuarlo como un módulo que verifica la correctitud del programa. Para apoyar esta buena práctica conviene contar con una función que convierta su argumento en una hilera usando los operadores de flujos: este es el trabajo que realiza el método estático TestCase::toString(), que usa un flujo básico como intermediario para producir la hilera resultado. Como este es un método estático no se poluciona el espacio de nombres del programador cliente; otra forma menos elegante de lograr lo mismo es usar un nuevo "*namespace*" para lograr lo mismo.

Para simplificar BUnit no se hace diferencia entre un caso de prueba de prueba exitoso y uno que no tiene éxito porque no se ha levantado la excepción adecuada. Esto contrasta con JUnit, que llama "falla" a un caso de prueba que ha levantado la excepción equivocada. Sin embargo, sí es posible usar BUnit para registrar que las excepciones han sido manejadas adecuadamente, por lo que no hay que incluir código especial para el manejos de bloques "*try* ".

8. CONCLUSIONES

Doxygen es una herramienta que es fácil usar. Si se toma un archivo de configuración similar al final de este artículo y se examinan algunos ejemplos específicos, es posible obtener automáticamente una documentación adecuada para programas y módulos. Explicaciones y contorsiones complicadas no necesarias al usar Doxygen junto con BUnit.

Al aplicar la técnica descrita en este artículo es posible incorporar en la especificación ejemplos de uso tomados del programa de prueba unitaria. La técnica consiste en definir un bloque de código ejemplo usando marcadores Doxygen "\dontinclude", "\skipline" y "\until" como se muestra en la Figura 4, en el que el código de ejemplo está encerrado en un bloque de corchetes dobles "{{ ... }}". Este bloque debe estar identificado por una hilera única que marca la prueba. La hilera de identificación se construye al concatenar la palabra "test" con el nombre del método a prueba: "test::connected()".

Los ejemplos de uso que acompañan a este documento son suficientemente completos para que quien los examine mejore la especificación de módulos incorporándoles los datos de prueba reales, tomados de programas de prueba funcionales. Esta mejora es sustancial si el ambiente de trabajo es uno en el que la documentación es una prioridad secundaria o no existe. Así se logra que la prueba unitaria de programas también sea un vehículo para aprender a mejorar la calidad de los programas.

Muchas herramientas de la familia xUnit son más completas que BUnit. Sin embargo, dado su diseño simple, BUnit tiene la ventaja de que se puede comenzar a usar inmediatamente sin necesidad de aprender mucho detalle. Debido a que los verbos usados en BUnit son similares a los de JUnit, el aprendizaje de nuevas herramientas similares se facilita mucho, por lo que BUnit puede ser una alternativa viable para introducir en un ambiente

profesional el uso de este tipo de herramienta. Siempre es muy importante que el diseño sea sencillo (diseñar es definir qué es lo importante, no incluir siempre todas las posibles opciones).

Receta:
(1) Verifique la documentación generada por Doxygen.
(2) Refine las pruebas que deben aparecer en la documentación.
(3) Mejore las pruebas para que sean completas o exhaustiva.

9. RECONOCIMIENTOS

Alejandro Di Mare aportó varias observaciones y sugerencias importantes para mejorar este trabajo.

10. CODIGO FUENTE

BUnit.zip: Todos los fuentes [.zip]
http://www.di-mare.com/adolfo/p/BUnit/BUnit.zip
BUnit.h: Documentación general
http://www.di-mare.com/adolfo/p/BUnit/index.html
test_BUnit.cpp: Prueba BUnit
http://www.di-mare.com/adolfo/p/BUnit/classtest__BUnit.html
TestCase: Cada caso de prueba es una instancia derivada de esta clase abstracta
http://www.di-mare.com/adolfo/p/BUnit/classTestCase.html
TestSuite: Colección de pruebas
http://www.di-mare.com/adolfo/p/BUnit/classTestSuite.html
test0.cpp: Ejemplo mínimo de uso de BUnit
http://www.di-mare.com/adolfo/p/BUnit/classtest0.html
test1.cpp: Muestra de uso de assertTrue_Msg()
http://www.di-mare.com/adolfo/p/BUnit/test1.cpp
rational<INT>: Operaciones aritméticas para números racionales
http://www.di-mare.com/adolfo/p/BUnit/classrational.html
test_rational.cpp: Prueba rational<INT>
http://www.di-mare.com/adolfo/p/BUnit/classtest__rational.html
ADH_Graph: Versión muy simplificada de un grafo
http://www.di-mare.com/adolfo/p/BUnit/classADH_1_1Graph.html
test_Graph.cpp: Prueba Graph
http://www.di-mare.com/adolfo/p/BUnit/classADH_1_1test__Graph.html
Disponibilidad Doxygen:
ftp://ftp.stack.nl/pub/users/dimitri/doxygen-1.5.4-setup.exe

Versión HTML de este artículo:
`http://www.di-mare.com/adolfo/p/BUnit.htm`

11. REFERENCES

[1] Allison, Chuck: The Simplest Automated Unit Test Framework That Could Possibly Work, C/C++ Users Journal, 18, No.9, Sep 2000. http://www.ddj.com/cpp/184401279.

[2] Beck, Kent: eXtreme Programming Explained, Addison Wesley, Reading, MA, USA, 1999.

[3] Di Mare, Adolfo: ¡No se le meta al Rep!, Reporte Técnico ECCI-2007-01, Escuela de Ciencias de la Computación e Informática, Universidad de Costa Rica, 2007. http://www.di-mare.com/adolfo/p/Rep.htm

[4] Di Mare, Adolfo: Uso de Doxygen para especificar módulos y programas, Reporte Técnico ECCI-2007-02, Escuela de Ciencias de la Computación e Informática, Universidad de Costa Rica, 2007. http://www.di-mare.com/adolfo/p/Doxygen.htm

[5] Fowler, Martin: Refactoring: Improving the Design of Existing code, Addison-Wesley Co., Inc, Reading, MA, 3rd printing Nov 1999. http://www.refactoring.com.

[6] Martin, James: Fourth Generation Languages, Prentice-Hall, ISBN 978-0133297492, 1986.

[7] van Heesch, Dimitri: Doxygen, 2005. http://www.doxygen.org.

[8] Llopis, Noel: Exploring the C++ Unit Testing Framework Jungle, Games From Within, 2004. http://www.gamesfromwithin.com/articles/0412/000061.html

12. CONFIGURACION DOXYGEN

```
1.  PROJECT_NAME           = "Prueba de rational:"
2.  OUTPUT_LANGUAGE        = Spanish
3.  OUTPUT_DIRECTORY       = .
4.  GENERATE_LATEX         = NO
5.  GENERATE_MAN           = NO
6.  GENERATE_RTF           = NO
7.  CASE_SENSE_NAMES       = YES
8.  INPUT_ENCODING         = ISO-8859-1
9.  INPUT                  = rational.h test_rational.cpp
10. RECURSIVE              = NO
11. QUIET                  = YES
12. JAVADOC_AUTOBRIEF      = YES
13. EXTRACT_ALL            = YES
14. EXTRACT_PRIVATE        = YES
15. EXTRACT_STATIC         = YES
16. EXTRACT_LOCAL_CLASSES  = YES
17. INLINE_INHERITED_MEMB  = YES
18. SOURCE_BROWSER         = YES
19. INLINE_SOURCES         = NO
20. STRIP_CODE_COMMENTS    = NO
21. REFERENCED_BY_RELATION= NO
22. REFERENCES_RELATION    = NO
23. FULL_PATH_NAMES        = NO
24.
25. SORT_MEMBER_DOCS       = NO
26. SORT_BRIEF_DOCS        = NO
27. CLASS_DIAGRAMS         = YES
28.
29. ENABLE_PREPROCESSING   = YES
30. MACRO_EXPANSION        = YES
31. EXPAND_ONLY_PREDEF     = YES
32. PREDEFINED             = "DOXYGEN_COMMENT" \
33.                          "Spanish_dox" \
34.                          "__cplusplus" \
35.                          "_MSC_VER=1300"
36. EXAMPLE_PATH           = .
37.
38. # Manual ==> http://www.doxygen.org/manual.html
```