ESPECIFICACIÓN DE MÓDULOS CON EJEMPLOS Y CASOS DE PRUEBA

MODULE SPECIFICATION USING EXAMPLES AND TEST CASES

Adolfo Di Mare

Universidad de Costa Rica, Escuela de Ciencias de la Computación e Informática, Costa Rica
adolfo.dimare@ecci.ucr.ac.cr

RESUMEN

Se plantea cómo aprovechar la formación del programador para que construya, por lo menos parcialmente, la especificación de sus módulos, incorporando en cada implementación datos de prueba que sean ejemplos de uso de cada rutina, con el fin de mejorar la reutilización de los módulos.

Palabras Clave: especificación, prueba unitaria de programas, ingeniería de sistemas, construcción de programas, implementación de programas.

ABSTRACT

It is explained how to take advantage of each programmer's education to have them construct, at least in part, module specifications, incorporating in each implementation test data that serve as usage examples for each routine, in order to improve module reuse.

KeyWords: specification, unit testing, software engineering, program construction, program implementation.

1. INTRODUCCIÓN

Especificar módulos para lograr reutilizarlos debiera ser el quehacer diario de todo programador [McConnell-2004]. A menos que el programa sea construido para ser ejecutado solo una vez, cada programador enfrenta la necesidad de dejar traza de su trabajo para que otros puedan modificar y mejorar lo que creó [Myers-2004]. Desafortunadamente, las habilidades que desarrollan los programadores no calzan con el trabajo de documentación que deben realizar para convertir cada módulo en un componente reutilizable. ¿Qué se puede hacer para remediar este problema? Una solución es contratar otras personas quienes se encarguen de escribir la documentación, anotando el código fuente con comentarios para que luego una herramienta como Doxygen [VH-2005] o Javadoc [Javadoc-2010] extraiga la firma, encabezado o prototipo de cada módulo y lo use para generar las páginas de documentación. Esta solución aumenta el costo de la construcción de los programas pues recurre a incrementar el rubro más caro en el desarrollo de sistemas: la planilla del equipo de programación. Es fácil argumentar que no hay otra solución, pues ya es difícil y caro entrenar programadores, por lo que obligar a que cada programador también tenga la capacidad de escribir especificaciones disminuiría la cantidad de programadores disponibles pues muchos no podrían adquirir las habilidades que les permitan redactar especificaciones, lo que en consecuencia haría más escasa la cantidad disponible de programadores con el consecuente aumento de costo.

La solución que en este trabajo se plantea es diferente. La idea principal es aprovechar la formación del programador para que construya, por lo menos parcialmente, la especificación de sus módulos, incorporando en cada implementación datos de prueba que sean ejemplos de uso de cada rutina, con el fin de mejorar la reutilización de los módulos.

2. REUTILIZACIÓN

Reutilizar significa no inventar de nuevo la rueda, aprovechando lo que otros hicieron para construir nuevas soluciones. No es un concepto nuevo, pues desde hace décadas se sabe cómo reutilizar componentes de programación [Kru-1992]:

1) Para que una técnica de reutilización sea efectiva debe reducir la distancia cognoscitiva entre el concepto inicial de un sistema y su implementación ejecutable final.

2) Para que una técnica de reutilización sea efectiva debe ser más fácil reutilizar los artefactos que desarrollarlos de nuevo desde el principio.

3) Para seleccionar un artefacto a ser reutilizado, es necesario saber qué es lo que hace.

4) Para reutilizar un artefacto de programación efectivamente es necesario encontrarlo más rápidamente de lo que se tardaría reconstruyéndolo.

La red Internet facilita mucho encontrar módulos en las bibliotecas. Por ejemplo, para saber cómo se usan los diccionarios **`map<>`** de la biblioteca estándar STL de C++ [Musser-2001], cualquiera de estas búsquedas produce la documentación requerida rápidamente:

- `http://search.yahoo.com/search?n=100&p=c%2B%2B+map`
- `http://www.google.com/search?num=100&q=c%2B%2B+map`

Esta forma de acceso también se obtiene si se usa una biblioteca que no es de uso público, pues ya existen motores de búsqueda local que permiten encontrar la documentación relevante a un problema específico en un gran repositorio (esta tecnología también tiene rato de haber sido puesta en práctica: [MBK-1991]). Estos motores de búsqueda e indexación ya están disponibles: lo que falta en muchos casos es la especificación de módulos, pues esta labor se elimina de los proyectos para recortar costos.

{1}	**template**< **class** T>
	void list<T>::insert(
	  list<T>::iterator p,
	  const value_type & v
	)

{2}	Agrega una copia del valor "v" al contenedor.
{3}	• El valor agregado queda antes de la posición "p" en el contenedor. • Si p==**this**->end() el valor queda al final de la lista. Precondición: <<< no tiene >>> Complejidad: O (**this**->size())

{4}	{{ // test::insert()
	  list<**long**> L = makeList_long("(3)");
	
	  L.push_front(2); assertTrue(L == makeList_long("(2 3)"));
	  L.push_front(1); assertTrue(L == makeList_long("(1 2 3)"));
	  L.push_back(4); assertTrue(L == makeList_long("(1 2 3 4)"));
	  L.push_back(5); assertTrue(L == makeList_long("(1 2 3 4 5)"));
	
	  L.insert(L.begin() , 0);
	  L.insert(L.end() , 6);
	  assertTrue(L == makeList_long("(0 1 2 3 4 5 6)"));
	
	  typename list<**long**>::iterator it = L.begin();
	  while (it != L.end() && (*it != 4)) { ++it; }
	  L.insert(it,4);
	  assertTrue(L == makeList_long("(0 1 2 3 4 4 5 6)"));
	}}

Figura 1: `list<T>::insert()`

3. ESPECIFICACIÓN

Dice un conocido adagio que quien no sabe adónde va llega a otro lado. También es bien sabido que es necesario apuntar antes de disparar. ¿Cómo se puede construir programas sin especificarlos primero? Sin la especificación, no es posible asegurar que se ha obtenido lo que se necesita. En la práctica se recurre al método de prueba y error para determinar si la funcionalidad de una pieza de programación es adecuada. Aunque esta estrategia pospone la etapa de documentación de los módulos construidos, muchas veces no sirve para lograr producir sus especificaciones correctas y completas porque al final del proyecto se recorta el presupuesto para documentar el código [Boehm-1981].

La especificación puede verse como un contrato en el que están definidos todos los servicios que la implementación del módulo es capaz de dar. La firma o prototipo del módulo siempre forma parte de la especificación, pues si no se definen los tipos y parámetros con que trabaja el módulo es imposible compilarlo o reutilizarlo. Algunos autores consideran que una especificación correcta debe estar escrita en un lenguaje formal, matemático. Pero otros sólo requieren de la especificación un alto rigor, con estas tres cualidades:

- Completa: debe decirse todo lo que hay que decir.
- Correcta: debe omitirse lo que no hay que decir.
- No ambigua

En pocas palabras, en la especificación no debe sobrar ni faltar nada. Debe estar toda la información esencial para implementar un módulo, y también para usarlo. Además, es conveniente que la especificación sea clara y sencilla de entender, de forma que para usarla el programador no requiera hacer un esfuerzo intelectual más grande de lo necesario [DiM-1999].

En la Figura 1 está una especificación bastante completa porque incluye las partes que generalmente forman parte de un buen sistema de documentación:

{ 1 } Encabezado, firma o prototipo.
{ 2 } Descripción corta de la funcionalidad.
{ 3 } Descripción detallada.
{ 4 } Ejemplo de uso.

{ 1 } El encabezado, firma o prototipo de un módulo contiene la información que necesita el compilador para traducir el programa. Incluye: (1) el tipo del valor retornado, (2) el tipo de los parámetros y (3) el nombre calificado del módulo. En el ejemplo de la Figura 1 el nombre calificado de la rutina es **`list<T>::insert()`** por lo que el compilador deduce que **`insert()`** es un método de la clase parametrizada **`list<T>`**. Como el valor retornado por **`insert()`** es **void** el compilador reconoce que este método no retorna ningún valor, que recibe una copia del objeto **`list::iterator<T>`** pero que, aunque recibe por referencia el parámetro "**`v`**", no puede modificarlo porque "**`v`**" es un parámetro **`const`**. El nombre calificado del tipo "**`value_type`**" es **`list<T>::value_type`** pues el lenguaje C++ permite usar abreviaciones en muchos contextos.

{ 2 } La descripción corta de la funcionalidad del módulo generalmente cabe en un solo renglón y dice qué hace el módulo (es un error decir cómo lo hace, pues eso es parte de la implementación). Usualmente con esta descripción se define o refleja la abstracción o esencia del componente de programación, pero debido a que es una descripción concisa usualmente no es suficientes para reutilizarlo. En el ejemplo de la Figura 1 la descripción corta dice que "**Agrega una copia del valor "v" al contenedor**".

{ 3 } La descripción detallada contiene la documentación que los programadores necesitan para usar el módulo. Ahí se indican todos los detalles y reglas que deben respetarse para que obtener los resultados correctos al usarlo. Este es el sitio en donde el programador indica cuáles son las pre-condiciones que deben cumplir los valores antes de usar el módulo y también deben quedar indicados cuáles son los resultado o post-condiciones. En el ejemplo de la Figura 1 es en la descripción detallada de la funcionalidad de la rutina en donde se aclara que la inserción se realiza antes de la posición "p", pues sin esta aclaración esta lista no funcionaría como lista sino que sería más bien un conjunto (**`std::set<>`**).

{ 4 } El ejemplo de uso le permite al programador apreciar inmediatamente la funcionalidad del módulo. También le permite copiar el texto del ejemplo para reutilizarlo inmediatamente. Cuando los programadores usan una biblioteca están acostumbrados a ponerle más atención a los ejemplos de uso que a la narrativa que define el comportamiento de cada módulo. Dice un conocido adagio que una imagen vale por mil palabras; en computación, un ejemplo vale más o menos eso mismo. Si la especificación no incluye ejemplos, en muchos casos los programadores encuentran difícil utilizarla.

Cualquier programador tiene el entrenamiento necesario para producir los ítemes **{1}** y **{2}** (prototipo + descripción corta). Para producir el ejemplo de uso **{4}** se requiere usar alguna herramienta que permita sirva para construir pruebas unitarias de módulos. En un contexto universitario conviene usar un módulo C++ similar a BUnit, que tiene la gran ventaja de que su implementación completa cabe en un único archivo de encabezado [DiM-2008c]. La implementación C++ BUnit.h es un sencillo marco de prueba unitaria de módulos basado en la herramienta JUnit [JUnit-2010] para Java [BG-1998].

4. IMPLEMENTACIÓN

Una implementación es un grupo de instrucciones imperativas a ser ejecutadas por el computador, escritas en uno o más lenguajes de programación. Cada implementación es una de las muchas posibles formas de escribir el código de un programa. Cuando se ha usado abstracción como técnica de diseño, cada implementación debe corresponder a alguna especificación pues a partir de la abstracción se llega a la implementación. Mientras con la especificación se define qué hace el programa, en la implementación queda escrito cómo se hace. Por eso, después de que ha sido definida la especificación, en general existen muchas posibles implementaciones diferentes que cumplen con la especificación. Optimizar una implementación significa mejorar el código para escoger una implementación que sea más eficiente.

¿Por qué los programadores no escriben especificaciones? Talvez una respuesta sea que la programación en muchos casos se ve como producción de código y se relega el diseño a un segundo plano. Es bien conocido que la aplicación de las modernas metodologías ágiles requiere de una buena definición previa de los requerimientos de la aplicación para luego pasar a la implementación; por una mala aplicación de la metodología puede llegarse a creer que el proceso solo requiere decidir en la mañana cuál es el código que hay que implementar en la tarde [Cohn-2004], estrategia exitosa solo si cada miembro del equipo de programación puede intuir cuáles son los requerimientos de la aplicación y de ahí deducir la especificación de cada módulo. Talvez esto es posible si hay que programar un sistema que requiere construir KxN pantallas para accesar una base de datos de N entidades, pero aún en este caso más adelante surgen problemas para integrar módulos que han sido producidos asincrónicamente y de manera aislada [McConnell-2004]. A veces los programadores son perezosos y ni siquiera escogen un buen identificador para cada módulo (algunos llegan al colmo de llamar "**`suma()`**" al módulo que "**`resta()`**", talvez porque la primera implementación que se les ocurrió comenzaba sumando), aunque el remedio lógico para evitar este inconveniente es que la organización siempre exija por lo menos una cantidad mínima de documentación para cada módulo.

La redacción de la descripción detallada de la funcionalidad de un módulo es la parte que requiere de una buena destreza en redacción (ítem **{3}** de la Figura 1). Si en el equipo de programación hay personas que puedan redactar las especificaciones detalladas **{3}**, se les responsabilizar de esta labor, pero en el caso contrario es necesario contratar personal técnico que pueda producir este tipo de documentación. En contraposición, a cualquier programador se le puede exigir que le dé formato a sus pruebas de manera que puedan ser utilizadas como ejemplo de uso **{4}**. Si se usa una herramienta de extracción y generación de documentación similar a Doxygen esta sencilla estrategia resulta en módulos que están mejor documentados. Estas herramientas de documentación producen la especificación de cada módulo, organizada apropiadamente, con base a las anotaciones y comentarios del código fuente en donde se define la funcionalidad y forma de uso de cada parte de la implementación. La documentación generada a partir del análisis del código anotado incluye no solo la firma o prototipo de cada módulo sino que también puede tener incorporados los ejemplos de uso que completan la especificación [DiM-2008a]. Por ejemplo, la especificación de la Figura 1 puede ser generada por Doxygen con base a las anotaciones del código fuente [DiM-2008c].

Es fácil argumentar que la implementación debe hacerse después de crear los casos de prueba, como lo explican los padres de la Programación Extrema [Beck-1999] cuando afirman que su estrategia para construir programas es "probar y luego codificar" [Beck-2002]. A quien le parezca demasiado extrema esta estrategia, se le puede convencer de usar una herramienta para prueba unitaria de programas similar a BUnit para lograr que cada programador se acostumbre a "diseñar y luego programar", pues es relativamente fácil usar el verbo “**`assertTrue()`**” para implementar algunos ejemplos de uso como casos de prueba. De esta manera también se evita desechar el código de prueba que escribe el programador cuando implementa un módulo (a los programadores les gusta codificar cada algoritmo junto con sus pruebas unitarias).

En algunas ocasiones la construcción de los casos de prueba de ejemplo puede facilitarse mucho si se usan funciones de ayuda. Por ejemplo, en la Figura 1 la rutina “**`makeList_long()`**” sirve para construir una lista a partir de una hilera que contiene los valores de la lista, lo que hace mucho

más simple implementar el caso de prueba de ejemplo. El uso de este tipo de andamiaje mejora mucho la calidad de los ejemplos porque permite hacerlos mucho más fáciles de entender.

```
/// Datos de prueba para \c insert() \c push_front() y \c push_back().
template <class T>
void test_list<T>::test_insert() {
    {{  // test::insert()
        list<long> L = makeList_long( "(3)" );

        L.push_front( 2 ); assertTrue( L == makeList_long( "(2 3)" ) );
        L.push_front( 1 ); assertTrue( L == makeList_long( "(1 2 3)" ) );
        L.push_back(  4 ); assertTrue( L == makeList_long( "(1 2 3 4)" ) );
        L.push_back(  5 ); assertTrue( L == makeList_long( "(1 2 3 4 5)" ) );

        L.insert( L.begin() , 0 );
        L.insert( L.end()   , 6 );
        assertTrue( L == makeList_long( "(0 1 2 3 4 5 6)" ) );

        typename list<long>::iterator it = L.begin();
        while ( it != L.end() && (*it != 4) ) { ++it; }
        L.insert(it,4);
        assertTrue( L == makeList_long( "(0 1 2 3 4 4 5 6)" ) );
    }}
    {   // Resto de las pruebas
        list<long> X = makeList_long( "(0 1)" );
        typename list::iterator it;  int i = 4;
        it = X.begin();
        X.insert(it, i);
        assertTrue( X.size() == 3 && ! X.empty() );
        i = 6;
        X.insert(it++, i);
        assertTrue( *it != 6 );
        assertTrue( X.size() == 4 );
    }
}
```

Figura 2: Implementación completa del caso de prueba para la especificación de la Figura 1

5. EXTRACCIÓN DE LOS EJEMPLOS DE USO

Para cada rutina debe existir un módulo de prueba. Por ejemplo, para clase **list<T>** es necesario crear el módulo de prueba **test_list.cpp** que es el sitio en donde están no solo los datos de prueba para **list<T>** sino también sus ejemplos de uso.

En la Figura 2 se muestra la implementación BUnit completa del método que contiene el caso de prueba de uso de **list<T>::insert()** de la Figura 1: después del bloque que contiene el ejemplo de uso encerrado entre corchetes dobles "**{{**" "**}}**" están las demás pruebas unitarias (hay pocas para ahorrar espacio).

Cada herramienta de documentación provee mecanismos diferentes para incluir código fuente en la documentación generada. Para hacer la extracción del código de ejemplo es necesario marcar el bloque de código que hay que extraer, indicando como mínimo el primer y el último renglón que hay que copiar del archivo que contiene la implementación. O sea, que la herramienta de documentación debe tener alguna rutina cuyo encabezado sería algo similar al siguiente

```
void extraiga(
    string archivo,
    string desde,
    string hasta
);
```

En la Figura 3 se muestra cómo se le indica a Doxygen que extraiga del archivo **test_list.cpp** la especificación del método **list<T>::insert()**. Estas anotaciones son comentarios C++ que indican que Doxygen debe analizar el código fuente contenido en el archivo **test_list.cpp** y extraer el texto que aparece entre el primer renglón en donde aparece la hilera "**test::insert()**" y el renglón subsiguiente que contiene "**}}**" (**test_list::test_insert()** es el método de prueba para la clase **list<T>**, como se ve en la Figura 2). El comando Doxygen **\dontinclude** sirve para definir el nombre del archivo de que se extraerá código para incluirlo en

la documentación: en este caso ese archivo es **`test_list.cpp`**. El comando **`\skipline`** sirve para definir cuál es la hilera que debe tener el primer renglón que será copiado y **`\until`** dice cuál es la hilera que debe contener el último renglón copiado. En este ejemplo, Doxygen copiará en la documentación generada desde "**`test::insert()`**" hasta "**`}}`**".

```
/** \fn template <class T>
        inline void list<T>::insert(iterator p, const value_type &v)

    \brief Agrega el valor \c "v" al contenedor.
    - El valor agregado queda antes de la posición \c "p" en el contenedor.
    - Si <code>p==this->end()</code> el valor queda al final de la lista.

    \pre <<< no tiene >>>
    \par Complejidad
    - O ( \c 1 )

    \dontinclude test_list.cpp
    \skipline    test::insert()
    \until       }}
    \see         test_list::test_insert()
*/
```

Figura 3: Comandos Doxygen que resultan en la especificación de la Figura 1

No hace falta que los ejemplos de uso hayan sido programados con BUnit si se usa Doxygen como herramienta para generar la documentación, pero sí es necesario encerrar el bloque de código que contiene la prueba en corchetes dobles "**`{{`**" "**`}}`**", pues es la hilera de llaves dobles la que marca el final del bloque de código que contiene el ejemplo de uso. Si se usa otra herramienta para generar la documentación, posiblemente el cambio más importante en el ejemplo sería no usar **`assertTrue()`** porque hay que usar un nombre es diferente, como **`Assert()`** o **`TOOL::assert()`**.

6. MEJORA DE LAS ESPECIFICACIONES

A primera vista parece muy difícil lograr mejorar la calidad de las especificaciones que los programadores producen, pero al usar cualquier herramienta es posible decorar con ejemplos de uso la información que de todas formas es necesaria para que el programa compile. La resistencia al cambio puede venir de algunos programadores que no quieran adoptar tecnologías nuevas, pero es sencillo darles una pequeña charla de inducción a su uso para lograr que puedan implementar hacer ejemplos de uso cada vez que escriben un módulo: de hecho, debiera ser suficiente mostrarles unos cuantos ejemplos similares al de la Figura 3, que no tiene gran complejidad. Es más sencillo comenzar con los programadores más nuevos quienes posiblemente después puedan comunicar a los otros programadores lo que han aprendido.

Una organización cuyos módulos computacionales no cuentan con especificaciones incurre en costos mayores cuando hay que hacerle modificaciones y mejoras a sus programas [Myers-2004]. Sin embargo, con un esfuerzo relativamente pequeño es posible mejorar sustancialmente la calidad de las especificaciones, sin llegar a la perfección que tienen algunas biblioteca de uso masivo como la biblioteca de Java o la STL adjunta al lenguaje C++.

7. CONTEXTO UNIVERSITARIO

La experiencia de usar ejemplos BUnit en un curso de la Universidad de Costa Rica [UCR] es muy positiva. A los alumnos no hace falta convencerlos de adoptar la técnica descrita en la Figura 3 porque siempre deben incluir documentación basada en pruebas unitarias en las soluciones a sus proyectos programados, pero el resultado colateral es que la calidad de la documentación mejora, lo que es luego apreciado por profesores de cursos posteriores. Esta es una estrategia que consiste en predicarle a los jóvenes para convencer a los viejos. El uso de las herramientas Doxgen primero [DiM-2008a] y BUnit después [DiM-2008c] se han propagado gracias a la aplicación de esta estrategia.

En la UCR en estos momentos se está trabajando en trasladar al ambiente Java/Javadoc la funcionalidad que se ha logrado con C++/Doxygen. Además, es necesario definir una estrategia que le permita a una organización incorporar ejemplos en la documentación de sus módulos. También se está trabajando en introducir herramientas que permitan darle formato al código pues se han hecho ya algunos proyectos usando Uncrustify [Uncrustify-2010], aunque algunos profesores prefieren usar herramientas que no son de uso libre pero que sí permiten forzar reglas de codificación de programas mediante el uso de plantillas de edición de texto.

CS1: Programación I
Objetivo: Proveer al estudiante la formación básica en programación para su adecuado desempeño en los cursos subsiguientes de la carrera, fomentándole sus habilidades generales para la resolución de problemas.
Contenidos: Algoritmos y estructuras de datos, bifurcación, iteración, recursividad, entrada y salida, clases y objetos, herencia, polimorfismo, jerarquías funcionales y procedimentales, excepciones, clases contenedoras, ordenamiento, concurrencia y sincronización, documentación, prueba de programas, herramientas, depuración.
Figura 4: Programación I

En la carrera de computación de la UCR se han realizado varios cambios en los 2 cursos de programación para mejorar la calidad de los programas producidos. En el curso de Programación I se usa la herramienta JUnit para permitirle a los estudiantes concentrarse en una pequeña parte del programa de manera que no se vean abrumados por la complejidad del lenguaje Java al principio del curso [DiM-2010c]. Esta exposición inicial a la plataforma de prueba unitaria JUnit no solo acelera el aprendizaje del lenguaje sino que permite introducir temas nuevos en el curso, como la programación concurrente [DiM-2010a].

CS2: Programación II
Objetivo: Introducir los fundamentos teóricos de programación para entrenar a cada estudiante en las técnicas básicas de construcción de programas, en especial en la especificación e implementación de módulos y artefactos de programación reutilizables, con el fin de que puedan usar herramientas de programación y participar en cualquier equipo dedicado a construir programas de mediana complejidad.
Contenidos: Generalidades, especificación, herencia y polimorfismo, parametrización de clases, árboles, validación de entrada de datos y manejo de excepciones, manejo de archivos planos, pruebas de programas.
Figura 5: Programación II

En el curso de Programación II desde hace varios años se usa C++ como lenguaje de implementación, pero recientemente el programa del curso fue modificado para que incluya el uso de pruebas unitarias además de concentrar más la atención en el uso de especificaciones para lograr la reutilización de módulos. Este segundo curso no consiste en hacer programas "más grandotes" en un lenguaje diferente a Java, sino que sirve para que los estudiantes aprecien mejor qué significa mejorar la calidad de los programas en el contexto de un lenguaje de riqueza semántica mayor a Java (Java es un lenguaje ideal para aprender a programar pero C++ es mejor para el resto de la carrera porque más expresivo y está más pegado a la máquina [DS-2008]).

8. CONCLUSIONES

La técnica que aquí se ha expuesto consiste en pedirle a los programadores que escriban un renglón que describa la funcionalidad del módulo (también hay que exigir que usen nombres significativos para cada módulo que implementen). Además, hay que instruirlos para que recodifiquen el código de prueba como un caso de prueba para alguna de las herramientas de prueba unitaria de programas. Luego es posible usar una herramienta de extracción de documentación para obtener a partir del código fuente especificaciones en las que, además de una descripción resumida de cada módulo, también aparece por lo menos un ejemplo de uso. Esta documentación generada también puede ser indexada por un motor de búsqueda que permita localizar rápidamente los módulos construidos con el fin de reutilizarlos: en computación, un ejemplo vale por mil líneas de código.

El truco que permite incluir datos de prueba en la especificación es muy simple: basta decorar la prueba con un nombre significativo y rodearla de corchetes o llaves dobles "**{{**" "**}}**"; luego es posible redactar la especificación detallada, pero esto es más difícil pues requiere de habilidades que muchas veces son extrañas a los programadores. Como esta técnica es tan simple, es difícil encontrar excusas para que los programadores no incorporen en sus implementaciones datos de prueba que complementen la especificación de cada módulo.

Receta:

{ 1 } Encabezado, firma o prototipo.
{ 2 } Descripción corta de la funcionalidad.
{ 4 } Ejemplo de uso "**{{**" "**}}**".

9. REFERENCIAS BIBLIOGRÁFICAS

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

[Beck-2002] Beck, Kent: Test driven development, Addison Wesley, Reading, MA, USA, 2002.

[BG-1998] Beck, Kent & Gamma, Erich: JUnit Test Infected: Programmers Love Writing Tests, Java Report, 3(7):37-50, 1998.

http://junit.sourceforge.net/doc/testinfected/testing.htm

[Boehm-1981] Boehm, Barry W.: Software Engineering Economics, (Prentice-Hall Advances in Computing Science & Technology Series), 1981.

[Cohn-2004] Cohn, Mike: User Stories Applied: For Agile Software Development, Addison Wesley 2004.

[DiM-1999] Di Mare, Adolfo: Reutilización de Contenedores Parametrizables con Lenguajes de Semántica Limitada, Tesis de Doctorado, Universidad Autónoma de Centro América [UACA], Costa Rica, 1999.

http://www.di-mare.com/adolfo/binder/index.htm

[DiM-2008a] Di Mare, Adolfo: Uso de Doxygen para especificar módulos y programas, I Congreso Internacional de Computación y Matemática, (CICMA-2008), celebrado del 21 al 23 de agosto en la Universidad Nacional (UNA), Costa Rica, 2008.

http://www.di-mare.com/adolfo/p/Doxygen.htm

[DiM-2008c] Di Mare, Adolfo: BUnit.h: Un módulo simple para aprender prueba unitaria de programas en C++ , X Simposio Internacional de Informática Educativa (ADH'08) realizado del 1 al 3 de octubre 2008, Salamanca, España, I.S.B.N.: 978-84-7800-312-9, pp425-430, octubre 2008.

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

http://www.di-mare.com/adolfo/p/BUnit-ADH-2008.pdf

http://www.di-mare.com/adolfo/p/BUnit-ADH-2008.pps

http://siie08.usal.es

[DiM-2010a] Di Mare, Adolfo: Introducción de la programación concurrente en el primer curso de programación, Artículo #12 de la Octava Conferencia del Latin American And Caribbean Consortium Of Engineering Institutions LACCEI-2010 (Consorcio de Escuelas de Ingeniería de Latinoamérica y del Caribe), realizado en la Universidad Católica de Santa María de Arequipa, Perú, junio 2010.

http://www.di-mare.com/adolfo/p/cs1cp.htm

http://www.laccei.org/LACCEI2010-Peru/Papers/Papers_pdf/CI012_DiMare.pdf

[DiM-2010c] Di Mare, Adolfo: Aprendizaje Java acelerado por casos de prueba JUnit, Artículo #22 del XVIII Congreso Iberoamericano de Educación Superior en Computación

[CIESC 2010] realizado en la Universidad Nacional de Asunción, Asunción, Paraguay, octubre 2010.

http://www.di-mare.com/adolfo/p/JUnit6d.htm

[DS-2008] Dewar, Robert B.K. & Schonberg, Edmond: Computer Science Education: Where Are the Software Engineers of Tomorrow?, The Journal of Defense of Software Engineering, enero 2008.

http://www.stsc.hill.af.mil/CrossTalk/2008/01/0801DewarSchonberg.html

[Javadoc-2010] Oracle: Javadoc Tool,

http://java.sun.com/j2se/javadoc/

[JUnit-2010] JUnit.org: Resources for Test Driven Development,

http://www.JUnit.org

[Kru-1992] Krueger, Charles W.: Software Reuse, ACM Computing Surveys, Vol.24 No.2, pp 131-183, Junio 1992.

http://portal.acm.org/citation.cfm?id=130856

http://www.biglever.com/papers/Krueger_AcmReuseSurvey.pdf

[McConnell-2004] McConnell, Steve: Code Complete: A Practical Handbook of Software Construction 2ed, MicrosoftPress, 2004.

[MBK-1991] Maarek, Yoëlle S. & Berry, Daniel M. & Kaiser, Gail E.: An Information Retrieval Approach for Automatically Constructing Software Libraries, IEEE Transactions on Software Engineering Vol 17, No. 8, pp 800-813, 1991.

http://se.uwaterloo.ca/~dberry/FTP_SITE/reprints.journals.conferences/MaarekBerryKaiser1991Libraries.pdf

[Musser-2001] Musser, David R.: STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library , ISBN 0-201-37923-6, Addison-Wesley, 2001.

http://www.cs.rpi.edu/~musser/stl-book/

[Myers-2004] Myers, Glenford J.: The Art of Software Testing 2nd Ed, John Wiley & Sons, Inc., 2004.

[Uncrustify-2010] Uncrustify: Source Code Beautifier for C, C++, C#, ObjectiveC, D, Java, Pawn and VALA, 2010.

http://uncrustify.sourceforge.net/

[VH-2005] van Heesch, Dimitri: Doxygen, 2005.

http://www.doxygen.org/index.html

10. CÓDIGO FUENTE

BUnit.zip: Todos los fuentes
• http://www.di-mare.com/adolfo/p/BUnit/BUnit.zip

BUnit.h: Documentación general
• http://www.di-mare.com/adolfo/p/BUnit/es/index.html
str2list.h: Documentación general [makeList_long()]
• http://www.di-mare.com/adolfo/p/str2list/es/index.html
• http://www.di-mare.com/adolfo/p/str2list/str2list.zip
test_BUnit.cpp: Prueba BUnit
• http://www.di-mare.com/adolfo/p/BUnit/es/classtest__BUnit.html
TestCase: Cada caso de prueba es una instancia derivada de esta clase abstracta
• http://www.di-mare.com/adolfo/p/BUnit/es/classTestCase.html
TestSuite: Colección de pruebas
• http://www.di-mare.com/adolfo/p/BUnit/es/classTestSuite.html
BUnit.h: General documentation
• http://www.di-mare.com/adolfo/p/BUnit/en/index.html
test_BUnit.cpp: Testing BUnit
• http://www.di-mare.com/adolfo/p/BUnit/en/classtest__BUnit.html
TestCase: Each test case is an instance derived from this abstract class
• http://www.di-mare.com/adolfo/p/BUnit/en/classTestCase.html
TestSuite: Test collection
• http://www.di-mare.com/adolfo/p/BUnit/en/classTestSuite.html

test0.cpp: Ejemplo mínimo de uso de BUnit
• http://www.di-mare.com/adolfo/p/BUnit/es/test0_8cpp_source.html
test1.cpp: Muestra de uso de assertTrue_Msg()
• http://www.di-mare.com/adolfo/p/BUnit/es/test1_8cpp_source.html

rational<INT>: Operaciones aritméticas para números racionales
• http://www.di-mare.com/adolfo/p/BUnit/es/classrational.html
test_rational.cpp: Prueba rational<INT>
• http://www.di-mare.com/adolfo/p/BUnit/es/classtest__rational.html

ADH_Graph: Versión muy simplificada de un grafo
• http://www.di-mare.com/adolfo/p/BUnit/es/classADH_1_1Graph.html
ADH_Graph_Lib.cpp: Funciones de apoyo para ADH_Graph.h
• http://www.di-mare.com/adolfo/p/BUnit/es/ADH__Graph__Lib_8h.html
• http://www.di-mare.com/adolfo/p/BUnit/es/ADH__Graph__Lib_8cpp.html
test_Graph.cpp: Prueba Graph
• http://www.di-mare.com/adolfo/p/BUnit/es/classADH_1_1test__Graph.html

Disponibilidad Doxygen:
• ftp://ftp.stack.nl/pub/users/dimitri/doxygen-1.7.1-setup.exe