Ediciones pensando el futuro
Editions thinking the future
Informática 2013
XV CONVENCIÓN Y FERIA INTERNACIONAL
15TH INTERNATIONAL CONVENTION AND FAIR
www.informaticahabana.com
www.informaticahabana.cu
ISBN: 978-959-7213-02-4

CONOCIMIENTO DE PROGRAMACIÓN MÍNIMO REQUERIDO PARA CONSTRUIR PROGRAMAS DE BUENA CALIDAD

MINIMUM KNOWLEDGE ON PROGRAMMING REQUIRED TO Write GOOD QUALITY PROGRAMS

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

RESUMEN: *Aunque muchas personas tienen los conocimientos básicos para escribir pequeños programas, no todos los programadores tienen los conocimientos de programación que se definen en este trabajo, los que son fundamentales para escribir programas de buena calidad.*

Palabras Clave: Abstracción, especificación, módulo, reutilización, prueba de programas, pruebas unitarias.

ABSTRACT: *Although many people have the basic knowledge to write small programs, not all developers have the programming skills defined in this paper, which are essential for writing good quality programs.*

KeyWords: Abstraction, specification, module, reuse, software testing, unit testing.

1. INTRODUCCIÓN

La calidad es un concepto relativo, pues depende del contexto en que se define. Por ejemplo, hoy en día ya no es necesario que un reloj de pulsera esté construido para que funcione durante siglos, pues muchos de esos aparatos se usan durante poco tiempo (a veces son sustituidos antes de un año). Otros objetos, como por ejemplo los guantes para lavar platos o las toallas húmedas de papel, están diseñados para tener una vida útil muy corta. En la industria de la computación, la calidad tiene un significado que también depende del contexto, pues si un programa se usará solo una vez no vale la pena invertir lo suficiente para que funcione durante esa única ejecución.

En lugar de elaborar un tratado teórico sobre lo que debiera saber un programador, el enfoque que aquí se usa es de ingeniería inversa, pues se destacan algunas carencias o defectos que deben ser corregidos en los programas, y con base a esa enumeración de deficiencias se define cuáles conocimientos debe tener un programador que produce software de calidad. Además, con el fin de lograr que esta definición de competencias del programador sea de utilidad práctica inmediata, en lugar de usar un concepto de calidad que contiene muchos atributos deseables, se ha escogido la ruta más directa al homologar el concepto de calidad con el de mejora y mantenimiento de programas. Esta simplificación del problema deja muchas áreas del control de calidad por fuera, pero tiene la ventaja de que hace posible definir los conceptos en el reducido espacio de un artículo académico como este.

2. ESPECIFICACIÓN

Sin especificación no hay calidad. Es fácil encontrar personajes ilustres que justifican el uso de especificaciones para construir programas [1],[2].

Sin embargo, no es difícil reconocer por qué es necesario hacer especificaciones si se usa la analogía del “hombre del rifle”: si uno dispara antes de apuntar, termina con una bala en el dedo gordo del pie. Si uno programa antes de especificar, termina programando otra cosa.

Ayuda mucho que cada módulo tenga una descrip-

ción corta que describa, en una frase, cuál es la funcionalidad implementada. A veces los programadores escogen nombres para cada rutina que oscurecen su función (como cuando llaman **sumador()** a un módulo que hace restas), pero lo usual es que el nombre del módulo tenga una descripción, talvez demasiado corta, de su funcionalidad. Lo especificación mínima es darle un nombre significativo a cada módulo y por eso siempre es necesario escoger buenos identificadores en cualquier programa.

```
long mcd();
```

La especificación de un módulo define su funcionalidad. En lugar de describir cuál es el algoritmo utilizado, más bien en la especificación se explica cuál es el cambio que resulta de la ejecución de la rutina. Este cambio resulta en la modificación de los valores almacenados en los parámetros, lo que implica que en cualquier especificación es necesario definir cuáles son los parámetros y en qué consiste la modificación de sus valores. Algunos módulos crean objetos nuevos, como ocurre con el que suma 2 matrices que debe retornar una nueva matriz, sin modificar sus argumentos. También es posible utilizar variables globales cuyo valor queda modificado como efecto colateral de la ejecución del módulo, pero generalmente se trata de evitar esta práctica porque es difícil lograr localizar cuáles variables globales son afectadas por algún módulo: de aquí nace la regla que dice "minimice el uso de variables globales".

```
long mcd( long a, long b);
```

Además de usar nombres adecuados para el módulo y sus parámetros, es importante incluir en la especificación al menos una frase corta que diga qué hace: "suma los valores de la lista", "encuentra el valor de corte", "calcula el valor de retorno", "encuentra el interés de referencia", etc. Esta descripción corta generalmente se incluye como un comentario que está en el encabezado del módulo.

```
// Calcula el Máximo Común Divisor
long mcd( long a, long b);
```

3. DOCUMENTACIÓN INTERNA

Sin documentación interna no hay calidad. Muchas son las razones por las que es necesario modificar un programa, pero generalmente la modificación se localiza en uno o varios módulos. Puede ocurrir que quien deba hacer la modificación sea la misma persona que construyó el módulo, pero si el sistema es de tamaño mediano o grande, lo más probable es que cada modificación sea llevada a cabo por una persona diferente.

Si la implementación no tiene documentación alguna, quien deba darle mantenimiento al módulo debe estudiar el código fuente para deducir, en un proceso de ingeniería inversa, cómo funciona. Por eso, también es importante decorar el algoritmo utilizado con comentarios cortos y concretos, que permitan después entender con mayor rapidez cómo funciona el módulo.

```
// Calcula el Máximo Común Divisor
long mcd( long a, long b ) {
    // solo trabaja con positivos
    long g = ( (a<0) ? -a : a );
    long r = ( (b<0) ? -b : b );
    long tmp;  // "r" es el resto

    do {
        tmp = r;
        r   = g%r; // módulo en común
        g   = tmp;
    } while  (0 != r);

    return  g;
}  // mcd()
```

4. ESPACIADO E INDENTACIÓN

Sin espaciado e indentación no hay calidad. A veces los programadores se ahorran algunas teclas al escribir sus programas, y terminan produciendo código ilegible. Afortunadamente, existen muchas herramientas que ayudan a embellecer el código fuente, como por ejemplo **Uncrustify**, [3] que funciona con lenguajes como C++ y sus derivados (C, Java, C#, etc.).

```
// Calcula el Máximo Común Divisor
long mcd(long a,long b){long g=((a<0)?-a
:a);long r=((b<0)?-b:b);long tmp;//resto
do {tmp=r;r=g%r;// módulo en común
g=tmp;}while (0!=r);return  g;}// mcd()
```

Es especialmente importante usar identificadores significativos para nombrar las variables y objetos del programa. Por eso, en lugar de `"chuleta24"` es conveniente llamar `"impuesto"` al porcentaje que se debe agregar al monto facturado: aunque el primer identificador es jocoso, la sonrisa se borra cuando haya que entender qué significa que la `"chuleta24"` en el contexto de un `"culantro"` queda multiplicada por el `"cangrejo"`.

5. DATOS DE PRUEBA

Sin datos de prueba no hay calidad. La falibilidad humana es inevitable: ningún programador puede

implementar sus programas sin cometer errores, aunque sean muy pequeños. Por eso, la prueba de programas inherente al proceso de su construcción. Desafortunadamente, bastantes programadores desechan las pruebas de sus programas por lo que, cuando hay que modificarlos, es necesario reconstruirlas nuevamente. Por eso, es importante que el código de prueba de cualquier módulo sea conservado para su reutilización posterior, cuando se modifica un programa.

La forma más simple de prueba de programas se logra probando, aisladamente, cada uno de sus módulos. A esta estrategia se le conoce como prueba unitaria (*unit testing*). Existen muchas bibliotecas especializadas en este tipo de pruebas. Algunas muy conocidas son las derivadas de JUnit [4] (*Java Unit Testing*), también conocidas como xUnit [5].

Es debatible cuál es el mínimo requerido para hacer pruebas [6], pero uno de los elementos primordiales de cualquier prueba es el verbo `assert()` [C arcaico] o **assertTrue()** [JUnit] que sirve para determinar si su argumento es o no verdadero. Una definición C++ mínima de esta importante herramienta es la siguiente:

```
/// Macro para prueba unitaria de módulos
#define  assertTrue(cond) do if  (!(cond))\
{ std::cout << "error[" << __LINE__      \
  << "] " #cond << std::endl; } whi le (0)
```

Si la prueba falla, un mensaje se error es emitido (en el flujo `std::cout` en este caso). Si la condición es verdadera, nada queda grabado. De esta manera, solo cuando hay fallas el programa de prueba produce algo (aunque sea paradójico, “nada” significa “todo funciona bien”).

```
assertTrue  ( 1 == mcd(1,2) );
assertTrue  ( 2*3*5 ==
    mcd( 2*2*2*2 * 3*3 * 5*5, 2*3*5 )
);
assertTrue  ( 30 == mcd( -3600,-30 ) );
```

Desde que fue inventada la programación ágil, en todas sus formas y facetas, sus proponentes siempre predicaron que al construir programas se hace primero la prueba y luego se implementa el algoritmo [4]. Debido a que el efecto de un módulo se refleja en los valores que quedan modificados después de su ejecución, las pruebas unitarias lo que deben verificar es si el resultado de la ejecución coincide con el valor esperado. Por eso, es natural que al hacer pruebas se use **assertTrue()** para determinar hacer esta comparación. En muchos casos basta hacer pruebas usando únicamente **assertTr ue()** , pero si los programas son muy complicados es necesario usar herramientas más completas.

6. ¡NO SE LE META AL RE P!

Si se le mete al *Rep* no hay calidad. Además de las construcciones básicas de la programación estructurada, los lenguajes modernos permiten usar programación orientada a los objetos, en los que es posible usar clases junto con herencia y polimorfismo para darle estructura a los valores usados en los programas. Las clases permiten organizar varios valores que cumplen un objetivo común, para lo que deben respetar algunas condiciones de estado llamadas la invariante de la clase. Debido a la fragilidad inherente de las clases, su parte interna, que también se conocer como su representación priva (***Rep***), debe estar protegida para evitar que quienes usen la clase hagan cambios que dejen valores inválidos o incorrectos.

```
class  rational { long  num, long  den };
```

Por ejemplo, un número racional es una clase en la que están almacenados 2 valores numéricos juntos: el numerador y el denominador. La invariante de la clase “`rational`” debe incluir una condición necesaria para que el valor almacenado sea válido: el denominador nunca debe ser cero (es incorrecto dividir por cero). Sin embargo, si los dos campos del número racional no están protegidos, un programador puede alterarlos dejando almacenado un valor incorrecto. Por eso los lenguajes modernos permiten usar ocultamiento de datos, de manera que sea posible controla el acceso a los campos de la clase (si son privados no son de libre acceso).

```
rational r;
{
    r.num = 23; // ¡ se le mete al Rep !
    r.den =  0; // no NO !
}
```

Los programadores novatos muchas veces se le meten a *Rep* porque no saben cómo evitar los errores de compilación que emite el compilador cuando usan directamente los campos de la clase. Afortunadamente, es sencillo declarar algunos métodos para la clase que permitan extraer los valores que se necesiten sin accederlos directamente. Los métodos mutadores [`set()`] sí permiten cambiar el valor almacenado en la instancia de la clase, mientras que los observadores [`get()`] sirven para leer valores sin hacer cambios a la clase.

```
// Convierte a (int ) el valor almacenado
long  rational::toInt() const  {
    return  (r.num/r.den);
} // método observador
```

La invariante de una clase no solo sirve para definir cuáles son los valores correctos para la clase sino que también permite implementar con mayor eficiencia los programas. Por ejemplo, si el número racional siempre está simplificado, pues el máximo común divisor entre el numerado y denominador es [1], para determinar si dos racionales son iguales no hace falta hacer ninguna multiplicación, pues nunca puede ocurrir que el valor [2/3] esté almacenado como [22/33] o [-200/-300].

7. EJEMPLOS DE USO

Los ejemplos de uso mejoran la calidad. Debido a que siempre es necesario hacer pruebas unitarias para cualquier módulo, conviene aprovechar esas mismas pruebas para complementar la especificación.

```
// Calcula el Máximo Común Divisor
long  mcd( long  a, long  b);
{{  //  test::mcd()
    assertTrue  ( 1 == mcd(1,2) );
    assertTrue  ( 2*3*5 ==
        mcd( 2*2*2*2 * 3*3 * 5*5, 2*3*5 )
    );
    assertTrue  ( 30 == mcd( -3600,-30 ) );
}}
```

Una forma de adherir a cada especificación un ejemplo de uso extraído de los datos de prueba es marcar el principio y final bloque de código relevante, como en ese ejemplo en que la marca inicial es la hilera [test::mcd()] y la marca final es [}}].

8. DESCRIPCIÓN DETALLADA

Los programadores son expertos en el arte de implementar algoritmos, pero rara vez tienen el entrenamiento necesario para escribir descripciones detalladas que sirvan para delimitar la funcionalidad de un módulo.

```
bool graph::connected(
    string src, string & dst ,
    list<string> & C
)
```

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** " si el camino existe, y "**fa l se**" en caso contrario.

La lista "**C**" contiene la secuencia de nodos del camino.

Si no hay camino, la lista "**C**" queda vacía.

Muchos programadores pueden escribir la descripción corta para el método graph::connected() [Calcula el camino desde "**src** " hasta "**dst** "], pero no pueden redactar el detalle de esa misma especificación. La forma más simple de remediar esta carencia es contratar especialistas que complementen las especificaciones con base en los datos de prueba unitaria; así como las bailarinas “bailan y no programan”, los programadores “programan y no redactan”.

9. RECOMENDACIONES

Una creencia popular dice que el cerebro humano no es capaz de lidiar con más de 7 asuntos simultáneamente [7], que es la cantidad de recomendaciones que se han justificados en este artículo:

[1] Minimice el uso de variables globales
[2] Use identificadores significativos
[3] Incluya la descripción corta en la especificación
[4] Use documentación interna
[5] Respete las convenciones de espaciado y de indentación
[6] Complemente con datos de prueba cada la especificación
[7] ¡No se le meta al *Rep*!

10. CONCLUSIONES

Justificaciones más extensas a las recomendaciones aportadas en la sección anterior se pueden encontrar en la literatura (por ejemplo [8],[9] o [10]), pero lo importante aquí es que estas reglas definen el mínimo requerido para que el programa sea de calidad. Encima de esto se puede agregar más, pero siempre ayuda mucho conocer cuál es el núcleo de conocimiento requerido para que una destreza esté bien desarrollada y ayuda a producir resultados adecuados.

11. AGRADECIMIENTOS

Alejandro Di Mare aportó varias sugerencias para exponer las ideas expresadas en este artículo. Además, varios docentes de la Escuela de Ciencias de la Computación e Informática también hicieron observaciones relevantes.

12. REFERENCIAS BIBLIOGRÁFICAS

1. Meszaros, Gerard: "xUnit Test Patterns - Refactoring Test Code" , Addison Wesley, 2007.

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

3. Uncrustify: Source Code Beautifier for C, C++, C#, ObjectiveC, D, Java, Pawn and VALA. http://uncrustify.sourceforge.net/

4. Beck, Kent && Gamma, Erich: "JUnit test infected: Programmers love writing tests", Java Report, 1998
http://junit.sourceforge.net/doc/testinfected/testing.htm

5. Hamill, Paul: "The xUnit Family of Unit Test Frameworks" O'Reilly, 2004.

6. Allison, Chuck: "The Simplest Automated Unit Test Framework That Could Possibly Work", C/C++ Users Journal Vol.18 No.9, Sep-2000, pp48-61.
http://www.drdobbs.com/184401279

7. Schenkman, Lauren: "In the Brain, Seven Is A Magic Number", ABC news Incide Science, 2009.
http://abcnews.go.com/story?id=9189664

8. Di Mare, Adolfo: “Especificación de módulos con ejemplos y casos de prueba”, Artículo CAL002 del V Taller de Calidad en las Tecnologías de la Información y las Comunicaciones (TCTIC-2011), realizado del 7 al 11 de febrero de 2011 en el Palacio de Convenciones de la Habana, en la Habana, Cuba.
http://www.di-mare.com/adolfo/p/BUnitXP.htm

9. 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

10. Sun Microsystems, Inc.: "Java Code Conventions”, 1997.

ACERCA DEL AUTOR

Adolfo Di Mare: Investigador costarricense en la Escuela de Ciencias de la Computación e Informática [ECCI] de la Universidad de Costa Rica [UCR], en donde ostenta el rango de Profesor Catedrático. Trabaja en las tecnologías de Programación e Internet. También es Catedrático de la Universidad Autónoma de Centro América [UACA]. Obtuvo la Licenciatura en la Universidad de Costa Rica, la Maestría en Ciencias en la Universidad de California, Los Angeles [UCLA], y el Doctorado (Ph.D.) en la Universidad Autónoma de Centro América.