Programación Razonada

K.Broda, S.Eisenbach, H.Khoshnevisan, S.Vickers*

1. Introducción

1.1. ¿Cómo saber si un programa hace lo que queremos?

Los programas se escriben para lograr que el computador haga algo para nosotros, así que es comprensible que si hemos escrito un programa, queramos tener cierta confianza en que el programa hace lo que pretendíamos. Una actitud frecuente es simplemente ejecutarlo y mirar. Si realiza algo inesperado, tratamos de corregir lo errores. (En inglés se les suele llamar 'bugs' (chinches), como si el programa hubiera cogido alguna infección incapacitante. Pero seamos francos y llamémoslos 'errores'.) Desafortunadamente, como Edgser Dijkstra apuntó, la comprobación (*testing*) sólo puede establecer la *presencia* de errores, no su ausencia, y es normal considerar que los programas irremediablemente tienden a contener errores. Sería sencillo decir que la respuesta es bien simple: ¡No escribir ningún error! ¡Conseguir un programa correcto a la primera! Los programadores nóveles descubren en seguida cuan fatuo resulta esto, pero entonces caen en la trampa opuesta de no tomar precauciones para evitar los errores.

En la programación práctica existen diferentes técnicas diseñadas para combatir los errores. Algunas nos ayudan a escribir directamente programas sin errores, otras en cambio aspiran a detectar los errores pronto, cuando son más fáciles de corregir. Sin embargo, la idea fundamental es: *cuanto mejor se entienda qué es lo que se supone que el programa debe hacer, más fácil será escribirlo correctamente.*

1.2. ¿Por qué preocuparse?

He aquí la garantía de un famoso y prestigioso sistema operativo:

> El proveedor rechaza cualquier garantía o responsabilidad, implícita o explícita, con respecto a este software, su calidad, funcionamiento, comercialización, o adecuación para un propósito particular. Como resultado, este software se vende 'tal cual' y el comprador asume por completo el riesgo en cuanto a su calidad y funcionamiento.

Afortunadamente, los programadores encargados de escribir el software no tomaron este descargo legal como la declaración definitiva de lo que se suponía debía hacer el programa. Trabajaron duro y concienzudamente para producir un producto bien pensado y útil, del cual poderse sentir orgullosos. Sin embargo, el más mínimo error en el software es potencialmente causante de un fallo catastrófico. Esto preocupa al departamento legal y, por las posibles consecuencias legales, procuran hacer todo lo posible para disociar a la compañía de los usos del software en el mundo real.

Hay otros contextos donde los litigios dejan de ser un factor teórico. Por ejemplo, si trabajamos en una compañía de software, nuestros compañeros pueden necesitar utilizar nuestro software, y querrán confíar en que funciona. Si algo va mal, los descargos legales no vienen a cuento. El director querrá saber qué es lo que fué mal y qué estamos haciendo al respecto.

Además, con frecuencia somos *nuestros propios clientes* cuando escribimos diferentes partes de un programa o reutilizamos partes de otros programas. Para cuando reutilizamos ese código, ya se nos ha olvidado qué es lo que hacía.

*Extraído y traducido por Yolanda Ortega-Mallén del libro *Reasoned Programming*, Prentice Hall Int., 1.994.

Resumiendo, la calidad que intentamos alcanzar en nuestro software, y la responsabilidad de evitar los errores, va más allá de lo que se pueda definir mediante obligaciones legales o contractuales.

1.3. ¿Qué *queríamos* que hiciera nuestro programa?

Nuestro software terminado contendrá un montón de *código* — instrucciones para el computador, escritas en algún lenguaje de programación. Es importante reconocer que la actividad que describe no tiene ningún sentido desde el punto de vista de los usuarios, porque ellos no necesitan saber qué es lo que ocurre dentro del computador. Esto sigue siendo cierto incluso para usuarios que sean capaces de leer y entender el código. Los usuarios están interesados en cuestiones como las siguientes:

- ¿Cuál es el efecto global del programa?
- ¿Es fácil entender lo que hace?
- ¿Es fácil de usar?
- ¿Ayuda a detectar y corregir los propios errores, o los encubre y luego nos castiga por ello?
- ¿Es rápido?
- ¿Cuánta memoria utiliza?
- ¿Contiene errores?

El código no expresa directamente nada de esto. Hablando en términos generales, la colección de instrucciones máquina en ella misma no dice nada acerca de lo que el programa realiza cuando se ejecuta en el mundo real.

Por lo tanto, al progresar desde nuestros imprecisos propósitos hacia el software completado, hemos realizado dos cosas distintas: primero, hemos convertido las ideas vagas en algo suficientemente preciso como para ser ejecutado por el computador; y segundo, hemos convertido las necesidades y requerimientos del usuario en algo totalmente diferente: instrucciones para el computador.

Nuestro único propósito es mostrar cómo dividir esta progresión en dos partes: primero, convertir las vaguedades en una relación precisa de las *necesidades y los deseos del usuario*; y después convertir ésta en instrucciones de computador.

La 'relación precisa de las necesidades y los deseos del usuario' se denomina *especificación*, y el punto crucial a entender es que expresa algo totalmente diferente del código, es decir, el interés del usuario en lugar de el del computador. Si la especificación y el código acaban expresando lo mismo en diferente forma — lo cual ocurre fácilmente si al especificar se piensa demasiado desde el punto de vista del computador — entonces hacer las dos cosas es una gran pérdida de tiempo.

1.4. Comportamiento local v.s. comportamiento global

Una distinción entre el código y la especificación es que mientras el código describe pasos individuales de ejecución — comportamiento *local* — la especificación se ocupa del comportamiento *global* general. A continuación un ejemplo (aunque no utiliza un lenguaje de programación ortodoxo).

PAP: Pasito A Pasito. El siguiente es un ejemplo de un programa PAP:

```
ANDAR 3 METROS; GIRAR A LA IZQUIERDA 90 GRADOS;
ANDAR 3 METROS; GIRAR A LA IZQUIERDA 90 GRADOS;
ANDAR 3 METROS; GIRAR A LA IZQUIERDA 90 GRADOS;
ANDAR 3 METROS; GIRAR A LA IZQUIERDA 90 GRADOS
```

El comportamiento *local* es dar cuatro paseos girando en ángulo recto cada vez; una propiedad *global* es que se termina en la posición inicial. El programa no describe esta propiedad global, tenemos que utilizar la geometría para deducirla. Esto no es trivial porque, con la geometría incorrecta, ¡la propiedad podría fallar! Consideremos este programa:

```
ANDAR 10000 METROS; GIRAR A LA IZQUIERDA 90 GRADOS;
ANDAR 10000 METROS; GIRAR A LA IZQUIERDA 90 GRADOS;
ANDAR 10000 METROS; GIRAR A LA IZQUIERDA 90 GRADOS;
ANDAR 10000 METROS; GIRAR A LA IZQUIERDA 90 GRADOS
```

Si empezáramos en el Polo Norte y anduviramos alrededor de la Tierra, no terminaríamos donde comenzamos.

El metro fué originalmente definido como una 10 millonésima parte de la circunferencia desde el Polo Norte hacia el Ecuador pasando por París. Así que el viaje descrito iría desde el Polo Norte a Libreville (vía París), después cerca de una pequeña isla llamada Nias, y de vuelta al Polo Norte, y después *hasta Libreville otra vez*. No volvemos al punto de partida. Así pués, las propiedades globales de un programa pueden depender mucho de supuestos geométricos ocultos: ¿Es nuestro mundo plano o redondo? No se explicita en el programa.

PAP no es un lenguaje de programación típico, y las propiedades de los programas no suelen depender de la geometría, aunque como PAP, su comportamiento pueda depender de factores del entorno. Sin embargo, sigue siendo cierto que el código de los programas usualmente describe tan sólo los pasos individuales de ejecución y cómo se ponen juntos, pero no su efecto global.

1.5. Programas razonados

Una vez que hemos precisado código y especificación, es un buen ejercicio tratar de compararlos todo lo precisamente que sea posible. Intentamos dar una precisión matemática no sólamente al propósito vago y general (obteniendo una especificación), sino también a todos los comentarios del programa. Pueden ser escritos en forma lógica, y se puede analizar de forma precisa si encajan o no con el código.

Cuando el código viene avalado por este tipo de especificación cuidadosa y de razonamiento, es un producto mucho más estable. Cuando lo hayamos escrito, tendremos mayor confianza en que funciona. Cuando lo reutilicemos, sabremos exactamente qué se supone que hace. Cuando lo modifiquemos, tendremos una idea más clara de cómo los cambios encajan en su estructura.

He aquí nuestro objetivo general:

$$\boxed{\text{especificación} + \text{razonamiento} + \text{código} = \textit{Programa Razonado}}$$

1.6. Programación razonada

Hemos presentado el Programa Razonado como el producto software final deseado, pero hay algo importante que decir acerca del proceso de su desarrollo, es decir, acerca de la *Programación Razonada*.

Podemos considerar como el propósito de una especificación el decir qué hace el código, pero éste es el camino incorrecto. Realmente, el propósito de el código es conseguir lo que la especificación ha descrito. Esto significa que es mucho mejor especificar primero y después codificar. En términos cotidianos, podemos realizar una tarea de forma más efectiva si en primer lugar entendemos qué es lo que estamos intentando obtener.

En palabras de Hamming:

> 'Teclear no sustituye a pensar.'

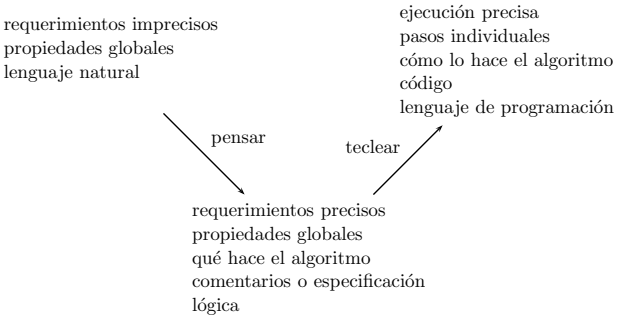
Esto significa que debemos precisar nuestras ideas antes de teclearlas — determinemos qué es lo que queremos antes de decirle al computador que lo haga.

Es siempre tentador ponerse directamente con el código. Ganamos nuestra experiencia inicial con programas muy cortos y vemos que el método funciona. Se le transmite algo al computador y nos devuelve rápidamente el resultado, lo cual resulta gratificante, incluso si usualmente muestra que hay errores. Muchos programadores continuan trabajando de esta manera a lo largo de toda su carrera. Consideran que, incluso si aceptan la idea de aspirar a Programas Razonados, todo ello es

un sueño imposible 'Sí, muy bien en teoría, pero...'. Escriben el código, y *entonces* — a lo mejor justo antes de la fecha de entrega — intentan clarificar sus razones.

Esto es un error. Hay dos aspectos esenciales del producto final que lo distinguen de las vagas intenciones iniciales: precisión y pasos locales de ejecución, y si vamos primero a por el código, estamos intentando obtener ambos aspectos a la vez. Por otra parte, si primero pensamos en la especificación, sólo buscamos precisión. Al fin y al cabo, es la especificación la que queda más cerca de las vagas intenciones originales, y no el código. Así que el primer paso debe ser siempre pensar cuidadosamente en nuestras intenciones y tratar de refinarlas a una especificación más precisa.

Después, el siguiente paso es convertir la globalidad (especificación) en localidad (código), y esto es mucho más fácil después del razonamiento inicial. De hecho, hay técnicas matemáticas específicas que realizan automáticamente parte de este proceso. Además, ligan la especificación y el código.



1.7. Módulos

La distinción que hemos hecho entre especificación y código, correspondiente a usuarios y computador, también tiene sentido dentro de un programa. Es corriente encontrar que parte del programa, con una tarea bien definida, puede hacerse bastante autocontenida y entonces se denomina *subprograma*, o *subrutina*, o *procedimiento* o *función*, o para partes más largas y estructuradas, *módulo*. La idea es que el programa total es algo compuesto, realizado *utilizando* componentes: por lo que el propio programa toma el rol de *usuario*.

Un módulo puede ser especificado y describe cómo su entorno, el resto del programa, puede llamarlo y qué realiza. La especificación describe todo lo que el resto del programa necesita conocer acerca del módulo. La implementación del módulo, el código que contiene, es su funcionamiento interno, está oculto y puede ser ignorado por el resto del programa.

La modularización es crucial para escribir programas grandes porque divide el problema total de codificación en subproblemas independientes. Una vez que se ha especificado un módulo, se puede codificar el interior olvidando el exterior, y viceversa. Las especificaciones de módulos actúan como *mamparas*, como las particiones en el casco de un barco que evitan que el agua de un agujero se extienda por todas partes, hundiendo el barco. Las especificaciones comparten el programa, de forma que si se detecta un error en un módulo se puede comprobar fácilmente si su corrección tiene alguna consecuencia para los otros. Esto evita el problema 'Hydra', de que al corregir un error se introduzcan otros diez errores nuevos.

1.8. Programando a gran escala

Estamos suponiendo algo significativamente simplista: que las *especificaciones se obtienen correctas a la primera*. Esto usualmente (aunque no siempre) es realista para programas pequeños,

por eso las técnicas presentadas se denominan programación *a pequeña escala*. La idea subyacente, de entender el punto de vista del usuario a través de una especificación, es también importante en programas a gran escala, pero las técnicas no pueden aplicarse directamente (primero especificar, después codificar). Para entender el por qué, debemos entender qué es lo que puede ir mal en una especificación.

La comprobación última — de hecho la definición — de *calidad* del software es la de adecuarse a su propósito. Se supone que la especificación captura formalmente esta idea de adecuación y, si se ha hecho bien, entonces un programa *correcto*, uno cuyo código satisfaga la especificación, será también uno de calidad. Si embargo, *las especificaciones pueden contener errores*, y esto se manifestará en inesperadas e indeseables construcciones en un programa formalmente correcto. Así pués, la *corrección* es sólo una aproximación a la *calidad*.

Ahora bien, hay muchas ventajas en olvidar la calidad y trabajar para la corrección. Por ejemplo, tenemos objetivos precisos (escribir el código para satisfacer la especificación) que son susceptibles de un análisis matemático, y podemos modularizar el programa para trabajar en la corrección de partes pequeñas y fáciles, olvidando aspectos más generales.

En la programación a gran escala, muchas de las técnicas prácticas que se usan están encaminadas a ayudar a corregir errores en la especificación lo más pronto posible, mientras todavía sea barato arreglarlos. Por ejemplo, la *validación de requerimientos* consiste en comunicar lo más efectivamente posible con los usuarios, para descubrir qué es lo que realmente necesitan y desean; después, numerosas metodologías de diseño nos ayudarán a obtener una buena especificación antes de empezar con la codificación; el prototipado produce código rápido y barato útil para descubrir aquellos fallos que se muestran mejor en una versión operativa (como la dificultad de uso en la práctica).

1.9. ¿Se puede demostrar la corrección de los programas?

Acabamos de distinguir entre calidad y corrección, y de explicar como la 'corrección', conformación con la especificación, es sólo relativa: si la especificación es errónea (es decir, no es lo que el usuario quería) entonces, también lo será el código, aunque sea 'correcto'. Pero, al fin y al cabo ambos, especificación y código, son formales, así que existe la posibilidad de dar demostraciones formales de esta corrección relativa — se podría decir que éste es el objetivo de los *métodos formales* en el software de computadores.

Merece la pena apuntar que lo que se verá son sólo 'métodos formales informales'. Hay dos razones importantes para ello.

La primera es que para dar una demostración formal de corrección necesitamos una *semántica* formal de nuestro lenguaje de programación, una relación matemática de lo que el programa realmente significa en relación a las especificaciones. Aunque daremos una semántica axiomática para algunas de las construcciones de programación, en general confiaremos en la comprensión informal de lo que significan.

La segunda es que el verdadero razonamiento formal tiene que incluir hasta el último detalle. Esto puede estar muy bien si es un computador (mediante una herramienta software) el que está comprobando el razonamiento, pero para los humanos tal razonamiento es tedioso hasta el punto de ser impracticable. Incluso en matemáticas puras, las demostraciones son 'rigurosas' — hasta el punto de no dejar ninguna duda — pero no formales. Nuestro objetivo es presentar un razonamiento riguroso.

Ahora bien, incluso un razonamiento riguroso corre el riesgo de contener errores, así que, si no podemos proclamar una corrección matemática infalible, ¿para qué sirve todo esto? No parece que trabajemos en un Programa Razonado como una estructura libre de errores. Sin embargo, la estructura de un Programa Razonado, incluyendo su especificación y razonamiento, es mucho más estable que un Programa No Razonado, es decir, simplemente código. Tendremos una idea más clara de su funcionamiento, y ello nos ayuda en primer lugar a evitar errores y, cuando los errores se cuelan, a entender por qué los cometimos y cómo corregirlos.

2. Especificaciones

Un programador responsable quiere que el cliente quede totalmente satisfecho con el programa, así que sus objetivos generales son los mismos: ambos quieren ver al final un producto satisfactorio y útil. Sin embargo, existen algunas tensiones entre el deseo del programador de una tarea fácil, y el deseo del cliente de un programa potente y que haga de todo. Todo ésto es cuestión de dinero. Un programa más potente cuesta más de producir. El cliente debe equilibrar sus necesidades con respecto a su presupuesto y el programador debe ser capaz de mostrar claramente la diferencia entre las especificaciones más potentes y las menos.

En este sentido, la especificación representa parte de un contrato entre el programador y el cliente. El contrato completo diría 'el Programador implementará software para esta especificación, y el cliente pagará determinada cantidad de dinero'.

2.1. La especificación como contrato

PIJOTERO (el cliente) y MANITAS (el programador) ya han hecho antes negocios juntos, y normalmente se entienden bien.

> `Acto 1`
> PIJOTERO: ¿Puedes escribirme un programa para calcular
> la raiz de un numero real?
> MANITAS: ¿ Puedo suponer que es no-negativo?
> PIJOTERO: Si.
> MANITAS: De acuerdo, puedo hacerlo.
> [*Apreton de manos y salida*]

Ahora hay un pacto entre caballeros de que MANITAS escribirá un programa para calcular la raíz cuadrada. Pero el contrato implica algo más sutil: si PIJOTERO utiliza el programa, entonces, siempre que la entrada sea un número no negativo, la salida será su raíz cuadrada. Esto es un contrato porque implica derechos y obligaciones interdependientes que regulan el modo de uso del programa.

En primer lugar, la entrada debe ser no negativa. Esto es una obligación para PIJOTERO, pero un derecho para MANITAS, quien está facultado para esperar, en provecho de su implementación, que la entrada sea no negativa. Pero, entonces está obligado a calcular la raíz cuadrada, y PIJOTERO tiene el derecho de esperar que ésta sea la salida.

Una especificación como ésta puede dividirse en dos partes:

- La *pre-condición* es la condición que el usuario garantiza sobre la entrada. Por ejemplo, la entrada es un número no negativo.
- La *post-condición* es la condición que el programador garantiza sobre la salida; por ejemplo, la salida es la raíz cuadrada de la entrada.

Nótese la tensión subyacente: el cliente querrá pre-condiciones débiles (para que el programa funcione para entradas muy generales) y post-condiciones muy fuertes (para que calcule muchas y muy precisas respuestas). El programador, en cambio, querrá lo contrario. Habrá, por lo tanto, alguna clase de diálogo para ponerse de acuerdo en los términos del contrato.

Veamos una formalización de la especificación anterior:

$$\{x \geq 0\} \ \ \textbf{fun}\ rcuad\ (x : entero)\ \textbf{dev}\ (y : entero) \ \ \{y^2 = x\}$$

2.2. Especificaciones defensivas: ¿qué ocurre si la entrada es errónea?

La especificación anterior es relativamente cómoda para MANITAS porque no tiene que preocuparse acerca de la posibilidad de una entrada negativa. Dicha preocupación se ha traspasado a PIJOTERO, quien debe ser cuidadoso. Si, por error, pasa una entrada negativa, estaría fuera del

contrato y no se sabría lo que iba a ocurrir. Podría obtener una respuesta sensata o una respuesta insensata, o una respuesta aparentemente sensata pero errónea, o un mensaje de error, o un bucle infinito, o una caída del sistema, o cualquier catástrofe.

Esta preocupación para PIJOTERO es usualmente aceptable si el único modo en que éste utiliza el programa de la raíz cuadrada de MANITAS es llamándolo desde un programa propio. Tiene entonces que mirar en cada lugar de llamada, y convencerse de que nunca usará una entrada negativa. Así, a cambio de un poco de cuidado, la función *rcuad* puede ejecutarse eficientemente sin tener que comprobar la entrada cada vez.

Por otro lado, PIJOTERO puede pretender utilizar el programa en un lugar expuesto, donde cualquier entrada concebible puede ser suministrada. En ese caso, PIJOTERO preferirá una 'especificación defensiva' para una función que le proteja contra malos argumentos. Cuando MANITAS pregunta si puede asumir que la entrada es no negativa, MANITAS contesta: 'No. Si es negativa, para la ejecución e imprime un mensaje de error.'

$$\{\ \}\ \textbf{fun}\ \mathit{defrcuad}\ (x : \mathit{entero})\ \textbf{dev}\ (y : \mathit{entero})\ \ \{(x < 0 \Rightarrow \mathit{error}) \wedge (x \geq 0 \Rightarrow y^2 = x)\}$$

2.3. Una pequeña contrariedad: tolerancia a errores

`Acto 2`
MANITAS: No puedo calcular raices cuadradas exactas.
Tiene que haber una tolerancia a errores.
PIJOTERO: Pero los programas que acabo de desarrollar asumen que las raices *son* exactas. Me costara mucho modificarlos.
MANITAS: Lo siento, pero tendras que modificarlos.
PIJOTERO: Tendras noticias de mis abogados.
[*Salida tarifando*]

La historia tiene un final feliz. El departamento legal de MANITAS ha incluido, muy prudentemente, la siguiente clausula general de rechazo en su software:

> Este software puede hacer cualquier cosa, o no hacerla. Cualquier cosa que diga MANITAS acerca de él no es operativo.

PIJOTERO deja de pensar en términos legales, y negocia con MANITAS la siguiente especificación:

$$\{x \geq 0\}\ \textbf{fun}\ \mathit{rcuad}\ (x : \mathit{entero})\ \textbf{dev}\ (y : \mathit{entero})\ \ \{\mid y^2 - x \mid < \mathit{tolerancia}\}$$

donde *tolerancia* es un valor a negociar.

El que las especificaciones tengan que ser revisadas a la vista de los intentos de implementarlas, es algo bastante común. Es una contrariedad, pero ocurre, y debemos entender cómo tratarlo.

En este caso, la post-condición se ha debilitado en beneficio de MANITAS, y esto provoca un trabajo extra para PIJOTERO, quien tiene que mirar en cada lugar de llamada de *rcuad* y comprobar si su razonamiento todavía es válido con la especificación revisada. Si todavía funciona, PIJOTERO será feliz. Si no, PIJOTERO tendrá que modificar su programa y su razonamiento.

2.4. Otros cambios en el contrato

La tolerancia a errores era una *post-condición debilitada*. Otras posibilidades son las siguientes:

- *Pre-condición reforzada*: MANITAS decide que necesita asumir más para que su rutina funcione. De nuevo, PIJOTERO tiene que comprobar cada llamada para asegurarse de que las nuevas pre-condiciones se cumplen.
- *Pre-condición debilitada o post-condición reforzada*: ahora la especificación es mejor para PIJOTERO, así que no tiene que hacer ninguna comprobación. Esta vez es MANITAS el que debe comprobar si su rutina todavía satisface las nuevas condiciones y hacer las modificaciones necesarias.

En cualquier caso, vemos que cuando una especificación cambia, los programas tienen que ser comprobados para asegurar que todavía se adecuan a la especificación revisada. Esta comprobación es tediosa, pero rutinaria: a causa de la forma en que las especificaciones han dado estructura lógica al programa, sabemos exactamente qué partes del programa necesitamos examinar, y exactamente qué es lo que hay que comprobar.

2.5. Un pequeño desliz: raíces cuadradas positivas

De momento PIJOTERO y MANITAS han acordado una especificación para *rcuad* con tolerancia a errores.

`Acto 3, Escena 1`
PIJOTERO: Parece que el resultado de *rcuad* es siempre no negativo.
¿Es eso cierto?
MANITAS: [*mira el codigo*] Si.
PIJOTERO: Bien. Es bueno saberlo.
[*Salida*]

Así es como el código puede infiltrarse en la especificación. Si acordaran una nueva post-condición reforzada:

$$\{\mid y^2 - x \mid \leq \mathit{tolerancia} \wedge (y \geq 0)\}$$

sería mejor para PIJOTERO, y MANITAS no resultaría peor parado, puesto que su código lo hace en cualquier caso. Sin embargo, si dicha post-condición no llega a escribirse puede suceder lo siguiente:

`Acto 3, Escena 2`
[*Es tarde por la noche.* MANITAS *esta sentado delante de su terminal*]
MANITAS: ¡Eureka! Puedo hacer que *rcuad* ejecute un 0.2 % mas rapido, haciendo negativo su resultado.
[*Borra la vieja version de rcuad*]

`Acto 3, Escena 3`
PIJOTERO: Mis programas han dejado de repente de funcionar.
MANITAS: [*mira el codigo*] No es culpa mia. *rcuad* satisface su especificacion.
[*Salida*]

Esta clase de malentendidos es muy frecuente cuando uno es su propio cliente (cuando escribimos nuestros propios procedimientos). Es fácil asumir que se puede entender un programa sencillo simplemente mirando el código; pero ésto es peligroso. El código sólo puede contarnos lo que el computador hace, no lo que el resultado tenía que ser. Todo lo que se asume debe estar en la especificación.