# Diseño por Contrato: Desarrollo de Software Fiable

Pablo Sánchez
Departamento de Matemáticas, Estadística y Computación
Universidad de Cantabria (Santander, España)
p.sanchez@unican.es

**Resumen**

El presente documento contiene un breve presentación de la técnica de *Diseño por Contrato*, un método eficaz para desarrollar software libre de fallos (*bugs*) desde el inicio del desarrollo del mismo. En primer lugar se expone la motivación del *Diseño por Contrato*. A continuación, se describe en qué consiste el *Diseño por Contrato*. Por último se describe el caso del Ariane 5, uno de los más famosos casos de desastres software donde el uso del *Diseño por Contrato* hubiese permitido ahorrar unos 500 millones de dólares.

## 1. Introducción

Es muy frecuente en el diseño de aplicaciones software encontrarse con funciones y procedimientos que no tienen definido su comportamiento para todos los valores posibles de sus parámetro de entrada. Por ejemplo, dada la clase MatrizCuadrada de la Figura 1, si creamos un objeto m del tipo MatrizCuadrada con dimensión 3, la llamada m.set(27,27,12), que trataría de asignar el valor 12 al elemento (27,27) de dicha matriz, carecería de sentido, puesto que los índices están fuera de las dimensiones de la matriz. Se trata por tanto de una llamada errónea ante la cual podemos aplicar las técnicas de tratamiento de errores que se describen a continuación, entre otras.

### 1.1. La técnica del avestruz

La *técnica del avestruz* consiste en esconder la cabeza en la tierra e ignorar lo que pase alrededor. Podemos distinguir dos formas diferentes de aplicar la técnica de la avestruz: la forma *segura* y la forma *insegura*.

La forma segura se limita simplemente a no hacer nada, impidiendo además que se haga algo erróneo. Aplicando la forma segura de la técnica del avestruz,

```java
public class MatrizCuadrada {

  // Array bidimensional que representa la matriz
  int [][] matriz = null;

  /**
   * Crea una matriz cuadrada de la dimensión indicada
   * @param dimension La dimensión de la matriz a ser creada
   */
  public MatrizCuadrada(int dimension) {
     matriz = new int[dimension][dimension];
  } // Matriz

  /**
   * Asigna un valor al elemento (i,j) de la matriz
   * @param i Fila del elemento a ser asignado
   * @param j Columna del elemento a ser asignado
   * @param valor Valor a ser asignado al elemento (i, j)
   */
  public void set(int i, int j, int valor) {
     this.matriz[i][j] = valor;
  } // set

  /**
   * Devuelve el valor del elemento (i, j)
   * @param i La fila del elemento a ser devuelto
   * @param j La columna del elemento a ser devuelto
   * @return El valor del elemento en la posición (i, j)
   */
  public int get(int i, int j) {
     return matriz[i][j];
  } //get

  /**
   * Devuelve la dimensión de la matriz
   * @return La dimensión de la matriz, es decir, al ser
   *         cuadrada, su número de filas o columnas
   */
  public int getDimension() {
     return this.matriz.length;
  } // dimension

} // Matriz
```

Figura 1: Clase MatrizCuadrada

el método set comprobaría que los valores de los parámetros de entrada sean correctos (ver Figura 2). Si dichos parámetros careciesen de sentido, es decir, fuesen negativos o superiores a las dimensiones de la matriz, no se haría nada. El programa continuaría su ejecución y el fallo se manifestaría más adelante, si llega a manifestarse[1].

[1]La experiencia del autor es que estos fallos más tarde o más temprano acaban manifestándose

```
1 public void set(int i, int j, int valor) {
2   if ((0 <= i) && (i < matriz.length) &&
3         (0 <= j) && (j < matriz[0].length)) {
4         matriz[i][j] = valor;
5   } // if
6 } // set
```

Figura 2: Técnica del avestruz segura

La técnica del avestruz segura puede plantear además ciertos interrogantes irresolubles dependiendo del método al cual se aplique. Por ejemplo, en el caso del método set, queda bastante claro en qué consiste la acción de *enterrar la cabeza*. Pero en el caso del método get, necesitamos retornar algún valor. Dado que la llamada es incorrecta, ¿qué retornaríamos en este caso? En estos casos, se suele optar por retornar valores inocentes como 0 ó 1, por ser los valores neutros para la suma y la multiplicación respectivamente. No obstante, esta decisión es tan arbitraria como devolver 73, por decir un número al azar.

La forma insegura de la técnica del avestruz se limita simplemente a no hacer nada, sin ni siquiera tratar de impedir que se haga algo erróneo. A diferencia de la forma segura, la técnica del avestruz insegura no comprobaría la corrección de los valores de los parámetros. Esta técnica es la que se aplica en el método set de la Figura 1.

Dependiendo del lenguaje de programación utilizado, un método como el método set de la Figura 1 provocará un comportamiento distinto, aunque siempre erróneos. la única diferencia es cuanto tarda en manifestarse el error.

En un lenguaje con comprobación dinámica de errores (*runtime checking*) tal como Java, una ejecución del método set con valores erróneos para los parámetros i e j provocará que se genere una excepción del tipo IndexOutOfBoundsException, al intentar acceder a posiciones situadas más allá de los límites del vector.

En lenguajes sin comprobación dinámica de errores, tales como C/C++, el acceder a posiciones no válidas de un vector no genera directamente una excepción, aunque provocará que escribamos en una zona de memoria donde en realidad no deseamos escribir. Destacar que en estos casos no tenemos ninguna seguridad acerca de en qué zona de memoria estamos escribiendo. Podemos estar intentando acceder a una zona de memoria a la cual no tengamos acceso, por lo que el sistema operativo terminará automáticamente la ejecución de nuestro programa[2].

En caso de que si tengamos acceso a la zona de memoria, el programa continuará su ejecución, pero hemos de ser conscientes de que hemos modificado aleatoriamente nuestro propio programa. La experiencia del autor es que estos fallos, sobre todo en el caso de que no generen violaciones de segmento, son fuente de largos quebraderos de cabeza. El motivo es que estos fallos que no se detectan inmediatamente provocan que los programas fallen en puntos muy lejanos al origen real del fallo, por lo que la detección del origen del fallo puede

[2]Estas son las *violaciones de segmento* tan familiares a los programadores C/C++

```
1 public void set(int i, int j, int valor) {
2   if ((0 <= i) && (i < matriz.length) &&
3         (0 <= j) && (j < matriz[0].length)) {
4             matriz[i][j] = valor;
5   } else {
6      throw new IndexOutOfBoundsException();
7   } // if
8 } // set
```

Figura 3: Control de errores mediante lanzamiento de excepciones

ser bastante tediosa.

En conclusión, la *técnica del avestruz* permite obviar el problema, pero no es una solución al mismo, que se acabará manifestando más tarde o más temprano.

## 1.2. Lanzamiento de Excepciones

El lanzamiento de excepciones es similar a la técnica de la avestruz segura, pero más elegante. Tras detectar que los parámetros de entrada no tienen los valores adecuados, se lanza una excepción para informar de que se ha producido una situación excepcional o anómala. Es deber del método que ha invocado al método set con valores erróneos para sus parámetros solucionar situación anómala. Un ejemplo de esta técnica aplicada al método set de la clase MatrizCuadrada se muestra en la Figura 3.

Lanzar una excepción sólo tiene sentido cuando nos podemos recuperar de dicha excepción. Por ejemplo, cuando queremos guardar un fichero, es posible que no podamos hacerlo porque dicho fichero esté abierto y siendo utilizado por otra aplicación, que lo mantiene bloqueado. En este caso, el método que trata de escribir en dicho fichero podría lanzar una excepción informando de que el fichero está actualmente siendo utilizado por otra aplicación y no podemos por tanto escribir en él. El método que haya invocado al método que ha generado esta excepción la recogería e informaría al usuario de la situación, indicándole que debe cerrar las aplicaciones que estén usando el fichero donde quiere almacenar la información. Si el usuario acepta y cierra dichas aplicaciones, nos podremos recuperar de dicha situación anómala y escribir en el fichero inicialmente indicado. No obstante, no siempre nos podemos recuperar de una excepción.

En situaciones donde no nos podemos recuperar de una excepción no tiene mucho sentido lanzarlas, dado que tampoco tiene sentido tratarlas. Si lanzamos una excepción que no se puede tratar, el programa lo único que podrá hacer es abortar su ejecución informando de que se ha producido una cierta excepción. Al igual que en los casos anteriores, el programa falla, aunque de una forma más elegante, dado que falla informando de porque falla. No obstante, al autor

```
1 public void set(int i, int j, int valor) {
2   if ((0 <= i) && (i < matriz.length) &&
3         (0 <= j) && (j < matriz[0].length)) {
4             matriz[i][j] = valor;
5   } else {
6         System.out.println("ERROR: MatrizCuadrada.set : " +
7             "Índices (" + i + ", " + j + ") fuera de rango ");
8   } // if
9 } // set
```

Figura 4: Control de errores mediante impresión de mensajes en pantalla

está convencido de que no existe ninguna forma bella de fracasar[3].

Adviértase que el resultado de lanzar una excepción no tratable sería el mismo que en la técnica de la avestruz insegura en un lenguaje con comprobación dinámica de errores. En ambos casos el programa finaliza abruptamente indicando la razón y origen de su fallo. La única diferencia estriba en quién genera la excepción. En el caso de la comprobación dinámica de errores, es el propio lenguaje el que genera la excepción. En el caso del lanzamiento de excepciones, se trataría del programador. En este último caso, el programador incluso definir sus propias excepciones. Esto último lo único que haría sería añadir algo más de elegancia al fallo, pero no solucionarlo.

Una versión más ligera del lanzamiento de excepciones consiste en imprimir un mensaje por pantalla informando de que se ha producido una situación excepcional o errónea; a la vez que se deja que el programa continúe su ejecución (ver Figura 4). Esta opción es una opción intermedia muy utilizada por los alumnos de los primeros cursos de las asignaturas de programación; y es una mezcla de la técnica del avestruz (se deja el programa continuar) y del lanzamiento de excepciones (se informa de la situación anómala). En ningún caso se resuelve el problema, pues el fallo sigue existiendo y la ejecución correcta del programa para todos los casos posibles no está en absoluto garantizada. No obstante, el indicar que se ha producido una situación errónea por pantalla facilitará la detección del origen del fallo y por tanto la tarea de depuración del programa.

En conclusión, ninguna de las técnicas anteriores, ni la técnica del avestruz ni el lanzamiento de excepciones, permite solucionar el fallo generado. Las técnicas comentadas hasta ahora permiten gestionar el fallo de una u otra forma; pero en todos los casos el programa terminará de forma no deseada. Por tanto, la solución al problema parece clara. No se trata de detectar las situaciones anómalas, sino de no generarlas.

Si queremos que el programa termine su ejecución correctamente, no deberemos producir llamadas a métodos que generen excepciones de las cuales no podamos recuperarnos. Es decir, la única técnica efectiva para gestionar de forma adecuada un fallo es no producirlo. Por tanto nuestro problema se reduce a: ¿Cómo sabemos que una llamada a un método no generará un fallo? O di-

[3]El caso del Ariane 5, que se describirá más adelante, ilustra esta sentencia

cho de otra forma: ¿Cómo podemos saber cuándo una llamada a un método es correcta?

La siguiente sección explica la técnica de *diseño por contrato*, que se basa en definir las condiciones bajo las cuales es seguro invocar a un método, así como aquellas suposiciones que es lícito realizar una vez que el método ha terminado su ejecución.

## 2. Diseño por Contrato

El diseño por contrato es una técnica de diseño software introducida por Bertrand Meyer [Jézéquel and Meyer, 1997, Meyer, 2000, 2009], cuyo objetivo es aumentar la fiabilidad de los programas software mediante la especificación de *contratos*.

Un contrato, informalmente hablando, establece en qué condiciones es seguro realizar una llamada a un método, y qué es lícito considerar como cierto tras la ejecución de dicho método. Desde un punto de vista formal, un contrato de un método $M$ se define como una tripleta $\{P\}M\{Q\}$, donde $P$ y $Q$ reciben el nombre de *precondición* y *postcondición* respectivamente.

Tanto las precondiciones como las precondiciones son fórmulas lógicas, del tipo $(x > 0) \wedge (y > 0)$. El significado de un contrato es "siempre que se invoque al método $M$ de forma que la precondición se satisfaga, es decir, que $P$ sea verdadero, el método $M$ termina y a su finalización la postcondición $Q$ es verdadera".

Es decir, la precondición establece las condiciones que tiene que cumplir cualquier programa que invoque al método $M$; mientras que la postcondición indica a qué se compromete el método $M$ siempre y cuando su precondición se satisfaga. Como si de un contrato de la vida real se tratase, ambas partes se comprometen a algo. En adelante denominaremos al programa que invoca al método $M$ el *cliente*; y al método $M$ el *proveedor*. De esta forma, la precondición establece las garantías que debe ofrecer el cliente al proveedor, mientras que la postcondición establece las garantías que debe ofrecer el proveedor al cliente.

Debe advertirse que la definición de contrato establece que el método $M$ debe terminar en un estado que satisfaga la postcondición sólo en los casos en que la precondición se satisface. En caso de que dicha precondición no se satisfaga, el método $M$ no tiene porque terminar en un estado que satisfaga la postcondición, de hecho no tiene ni porque terminar. Esto significa que siempre que no se satisfaga la precondición, el método $M$ puede hacer lo que le venga en gana, porque al violar el cliente las condiciones del contrato, el contrato se anula. Es responsabilidad del cliente asegurar que cada petición al proveedor satisface las precondiciones del contrato.

Por ejemplo, para el método set de la clase MatrizCuadrada se podría definir el contrato que se muestra en la Figura 5. La precondición asegura que los índices del elemento a modificar estén dentro de las dimensiones de la matriz. La postcondición indica que si inmediatamente después de realizar una llamada al método set realizamos una llamada al método get con los mismo índices,

```
1 // Pre: ((0 <= i) AND (i < this.getDimension())) AND
2 //      ((0 <= j) AND (j < this.getDimension()))$
3 public void set(int i, int j, int valor) {
4    ..
5 } // set
6 // Post: valor == this.get(i,j)
```

Figura 5: Contrato para el método set de la clase MatrizCuadrada

```
1 public static void init(MatrizCuadrada m) {
2   for(int i=0;i<m.getDimension();i++) {
3       for(int j=0;j<m.getDimension();j++) {
4           m.set(i,j,0);
5       } // for
6   } // for
7 } // init
```

Figura 6: Código libre de llamadas erróneas

obtendremos el valor que acabamos de escribir.

El lector no obstante podría argumentar sino estaría de más verificar al comienzo de cada método que la precondición del mismo se satisface, tal como ocurre, por ejemplo, en el caso de la técnica de la avestruz segura (ver Figura 2). Antes de responder a esta pregunta, examinemos el código de la Figura 6. En este caso puede observarse que, por la propia construcción de los bucles, nunca se realizará una llamada con valores incorrectos para los parámetros i y j. Por tanto, si el método set comprobase que los valores de los parámetros son correctos, estaríamos haciendo un trabajo redundante e inútil. Ello nos lleva a enunciar el *principio de no redundancia*.

**Principio de No Redundancia**

Bajo ningún concepto debe el cuerpo de un método verificar el cumplimiento de la precondición de la rutina.

Por tanto, siguiendo un esquema de diseño por contrato, cada método debe declarar las precondiciones bajo las cuales tiene sentido o es factible que dicho método realice su trabajo y termine de forma que satisfaga su postcondición. Son los clientes de los métodos los que tienen que asegurar que no se realizan llamadas a dichos métodos que no satisfagan las precondiciones. Para ello deberán realizar todas las comprobaciones que se consideren necesarias. Por ejemplo, en el caso de la Figura 6, por la propia construcción del programa, no es necesario realizar comprobación alguna por parte del cliente.

En caso de que se realice una llamada a un método que no satisfaga las

```
1  public class CuentaBancaria {
2
3       protected double totalIngresos = 0.0;
4       protected double totalGastos   = 0.0;
5       protected double saldoActual   = 0.0;
6
7       /**
8        * Reintegra al dueño de la cuenta bancaria la cantidad
             indicada
9        * @param cantidad La cantidad a ser reintegrada
10       */
11      public void sacarDinero(double cantidad) {
12           this.totalGastos = this.totalGastos + cantidad;
13           this.saldoActual = this.saldoActual - cantidad;
14      } // sacarDinero
15
16 } // CuentaBancaria
```

Figura 7: Clase Cuenta Bancaria

precondiciones del mismo, se considerará como un *bug*. Pero el responsable del *bug* no será el método invocado (donde posiblemente se genere la excepción o se aborte la ejecución del programa), sino el cliente de dicho método.

Por tanto, de acuerdo con la técnica del diseño por contrato, las precondiciones de todos los métodos deben hacerse explícitas, de forma que los clientes de dichos métodos sepan bajo qué condiciones es seguro invocarlos. Es decir, hay que hacer todos los contratos explícitos.

# 3. Preservando la Consistencia: Invariantes

Junto con las precondiciones y las postcondiciones, un mecanismo muy útil para mantener la consistencia de una aplicación software es la definición de invariantes. Un *invariante* es una condición o predicado lógico que ha de ser verdadero durante todo el tiempo de ejecución de la aplicación software.

Por ejemplo, dada la clase CuentaBancaria de la Figura 7, un claro invariante de la misma es que el saldoActual debe ser igual a la diferencia entre el totalIngresos y el totalGastos. Por tanto, se podría definir como invariante el siguiente predicado: $\{saldoActual == (totalIngresos - totalGastos)\}$.

Cada rutina, procedimiento o método ejecutado por la aplicación software deberá encargarse de asegurar que no se viole el invariante. Por ejemplo, el método sacarDinero modifica el atributo totalGastos, por lo que si el invariante se satisfacía al comienzo del método, al modificar dicho atributo habremos violado dicho invariante. Por tanto, es responsabilidad del método restaurar el invariante antes de terminar su ejecución. Para ello, el método modifica el valor del atributo saldoActual, de forma que se satisfaga de nuevo el invariante. Obsérvese que entre la primera y la segunda sentencia del método sacarDinero, el invariante es temporalmente inválido. Ello es natural, pues no podemos ejecutar varias

sentencias al mismo tiempo, o aunque podamos, no hay una razón de peso para hacerlo.

En general, el orientación a objetos se considera que es perfectamente válido violar el invariante durante la ejecución de un método público, siempre y cuando se garantice que el invariante se satisface tanto al comienzo como al final de la llamada a dicho método. Por tanto, al igual que con las precondiciones, el cliente debe asegurarse de que todos los invariantes de la aplicación se satisfacen antes de realizar una llamada a un método; y como si de una postcondición se tratase, el cuerpo de un método debe asegurarse de que el invariante se sigue satisfaciendo al final de su ejecución (aunque en medio de la ejecución se puede haber violado el invariante repetidas veces.)

# 4. La necesidad de hacer los contratos explícitos: El caso del Ariane 5

El 4 de Enero de 1996 se celebró el vuelo inaugural del Ariane 5, un cohete de la Agencia Espacial Europea y uno de sus productos estrellas en aquellos momentos. El vuelo tuvo una duración total de 40 segundos y acabó con la explosión del cohete, estimándose las pérdidas en medio billón de dólares [Jézéquel and Meyer, 1997], sin contabilizar el perjuicio que aquello supuso para la imagen y la credibilidad de la Agencia Espacial Europea.

Tras la consiguiente investigación, el informe técnico concluía que la explosión del Ariane 5 se debió a un fallo software. El fallo procedía de un módulo denominado SRI (*Inertial Reference System*). Tras la ignición inicial, se necesitaban realizar ciertos cómputos para ajustar el SRI. Dichos cómputos no consumían más de 9 segundos, y tras este periodo de tiempo, el SRI no servía para absolutamente nada. No obstante, por motivos técnicos, si el SRI se detenía, el proceso de reinicio, por ciertos motivos técnicos de bajo nivel, podía consumir varias horas. Por tanto, los ingenieros del Ariane decidieron no parar el SRI para que, en caso de que el despegue se abortase en el último momento por cualquier razón (como por ejemplo, porque apareciese en el cielo una densa y espontánea bandada de ánsares), no tener que esperar varias horas a que concluyese el proceso de reinicio del SRI para poder intentar un nuevo despegue. Hasta este punto, todo parecen decisiones correctas y acertadas y no se les puede reprochar nada a los ingenieros del Ariane 5.

El problema fue que el software SRI, tras los 9 segundos de trabajo útil, es decir, mientras realizaba cálculos que a nadie le interesaban, generó una excepción que causó que el software perdiese el control del cohete, provocando su explosión. La excepción que fue el origen de la explosión del Ariane 5 se generó porque se produjo un desbordamiento al convertir un flotante de 64 bits a un entero de 16 bits. El Ariane 5 explotó por tanto por una simple excepción del tipo *Arithmetic Overflow*, tan común entre los alumnos de un primer curso de programación. La excepción no fue recogida por ningún manejador, por lo que el software paró su ejecución, causando la explosión del cohete.

El módulo software del SRI se había reutilizado tal cual del proyecto Ariane 4. Ésta es también una decisión loable y acertada por parte de los ingenieros del Ariane 5. Si existe un módulo software que sirve para resolver nuestra tarea, y dicho módulo software además tenemos la garantía de que ha sido probado y funciona correctamente (el proyecto del Ariane 4 fue un éxito), lo lógico y lo recomendado es usarlo, en lugar de crear uno nuevo.

El motivo por el cual no existía manejador para dicha excepción es, que por ciertos motivos de seguridad, la carga de trabajo del procesador del SRI no podía ser superior al 80 %. Para mantener la carga de trabajo dentro los límites establecidos, sólo se crearon manejadores de excepción para las conversiones de variables que eran susceptibles de provocar desbordamientos aritméticos. Dada la trayectoria del Ariane 4, se podía asegurar con total certeza que ciertas conversiones no iban a provocar jamás desbordamientos, porque el rango de valores que podían recibir estas variables *nunca* iban a provocar desbordamientos. Por tanto, los ingenieros del Ariane 4 decidieron no crear manejadores de excepción para aquellas conversiones para las cuales sabían con absoluta certeza que eran seguras. Al eliminar dichos manejadores, aseguraban también que la ocupación del procesador del SRI se mantenía dentro del límite del 80 % establecido. Por tanto, la decisión de eliminar estos manejadores dentro del proyecto del Ariane 4 se debe considerar también como una decisión acertada. En todos los proyectos software es frecuente tomar decisiones de compromiso entre fiabilidad y eficiencia.

El verdadero problema fue que cuando los ingenieros del Ariane 5 reutilizaron el módulo software del SRI para el Ariane 4 desconocían las decisiones que habían tomado los ingenieros del Ariane 4 comentadas en el párrafo anterior. El Ariane 5 tenía una trayectoria de vuelo diferente a la del Ariane 4, por lo cual las conversiones de variables que eran seguras para el Ariane 4, es decir, pasaban a ser inseguras dada la trayectoria del Ariane 5. No obstante, los ingenieros del Ariane 5 desconocían este hecho puesto que no había sido hecho explícito en ninguna parte.

Por tanto, la verdadera causa del problema que originó la explosión del Ariane 5 fue que los ingenieros del Ariane 5 reutilizaron un módulo software del cual desconocían sus precondiciones. En tiempo de ejecución se produjo una llamada a dicho módulo que no satisfacía estas precondiciones, por lo que se produzco una excepción que causó la parada del sistema y la consiguiente explosión. Si las precondiciones del módulo reutilizado del Ariane 4 hubiesen estado disponibles, los ingenieros del Ariane 5 hubiesen podido comprobar que no se producían llamadas incorrectas a dicho módulo. Dada la trayectoria del Ariane 5, los ingenieros hubiesen detectado que ciertos parámetros, en el caso particular del Ariane 5, podían tomar valores incorrectos, realizando las acciones oportunas para prevenir dicho error.

Por tanto el verdadero fallo fue que los ingenieros del Ariane 4 no hicieron explícitas las precondiciones del módulo software para el SRI, y que los ingenieros del Ariane 5 reutilizaron dicho módulo software asumiendo que dichas precondiciones no existían.

Adviértase que si hubiese existido un manejador para la excepción que pro-

vocó el desastre del Ariane 5, la situación no habría sido muy diferente. Como hemos comentado, los manejadores de excepción tienen sentido cuando nos podemos recuperar de las situaciones excepcionales. En el caso del Ariane 5, el manejador de excepción podría haber hecho una de las siguientes acciones, tras recoger la excepción:

1. No hacer nada. El sistema no se hubiese parado, pero el SRI continuaría sus cálculos con valores incorrectos, que, más tarde o más temprano, conducirían posiblemente a un mal funcionamiento del SRI, haciendo que el sistema entrase en un estado de comportamiento aleatorio.

2. Parar la ejecución del programa mostrando un mensaje de error por pantalla. En este caso, se podría haber ahorrado el esfuerzo necesario realizado tras la explosión para encontrar el origen del fallo, pero el cohete hubiese explotado igualmente, que es lo que queríamos evitar.

Por tanto, si no podemos recuperarnos de las excepciones, no tiene sentido generarlas. No se trata de detectar excepciones, sino de no generarlas.

# 5. Sumario

La técnica del *Diseño por Contrato* permite delimitar las responsabilidades acerca de la comprobación de los valores que pueden tomar ciertas variables cuando se realizan llamadas a métodos o rutinas software. Siguiendo una filosofía de *Diseño por Contrato* se puede conocer cuando una llamada a un método o rutina es susceptible de provocar un error que comprometa la fiabilidad del sistema.

La filosofía de *Diseño por Contrato* permite además reducir el número de comprobaciones a realizar dentro de un programa software y que están destinadas a asegurar su robustez y fiabilidad. Un exceso de comprobaciones puede crear un sistema altamente robusto a la par que altamente ineficiente y difícil de mantener, ya que hasta la menor pieza de código tendría que asegurar la ausencia de un sinfín de exóticas situaciones de error; aún cuando por la propia construcción del software sea absolutamente imposible que dichas situaciones se produzcan. Por contra, una ausencia de comprobaciones generará sistemas escasamente robustos y poco fiables, que ante la menor variación en los datos de entrada presentará un comportamiento aleatorio.

La técnica de *Diseño por Contrato* permite definir claramente quién tiene que realizar dichas comprobaciones y cuando es necesario realizarlas.

# Referencias

Jean-Marc Jézéquel and Bertrand Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(1):129–130, 1997.

Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, March 2000.

Bertrand Meyer. *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer, September 2009.